MEKHOS REFERENCE MANUAL

M

THE TUTORIAL MACHINE ARCHITECTURE

M0 INTRODUCTION

M is a computer ARCHITECTURE designed to support the teaching of hardware and software, spanning the range from a first course in computer basics to final-year projects in compilers, concurrency and operating systems. M programs are executed by an EMULATOR, E, a program running on a real computer that interprets M instructions and imitates their behaviour. The M architecture; its emulator, E; loader, L; assembler, A; debugger, D; and OS kernel, K; together constitute the **mekhos** system. *mekhos* is the ancient Greek root of words such as 'machine' and 'mechanism'; it means 'means' (to an end), and so is appropriate for a project that is intended to be a means by which computer mechanisms can be better understood.

This document is the reference manual for the **mekhos** system, not a tutorial introduction to M.

The architecture of M attempts to satisfy a demanding set of requirements:

- It avoids the excessive complexity and irregularity of commodity architectures, by maximizing the orthogonal generality and symmetry of its instruction set.
- It has a *very* simple subset, suitable for use in a course on computer basics, but able to implement all the high-level language structures likely to be seen by students of such a course.
- It is, nevertheless, a fully-featured design, comparable in functionality with real-world products.
- It provides efficient and simple support for all standard data types and control structures.
- It is possible to write an efficient, portable and well-structured emulator, itself an object of study.

One architectural consequence of the above is that the design eschews the use of condition codes. Another is that some aspects of M are **implementation defined**: that is, left open, to allow efficient emulation on a variety of host architectures. §11 lists the definitions of these aspects for each current implementation.

A third consequence is that M is *not* a RISC architecture, although it does have a 'pure RISC' subset, and might reasonably be described as 'post-RISC'; it is certainly much simpler than many architectures that claim the cachet of RISC. M's most-used ALU operations have a **one-address** format, combining a stored or immediate operand with a register. This almost halves the number of instructions needed for expression evaluation, and makes short examples more focused and concise—an important pedagogical consideration. M also avoids making excessive demands for internal dataflow. In particular, no instruction writes back to more than two internal registers, and no instruction accesses more than two store operands, which are in adjacent locations. (In fact, the architectural style of M is rather in the classical British tradition of one-address, general-register machines that was initiated by Christopher Strachey with his revolutionary instruction set for the Ferranti Pegasus, and that later achieved great success in the ICL 1900 Series.)

A characteristic of M is the (almost) equal treatment of signed and unsigned integer arithmetic. There is also great symmetry between integer and floating point arithmetic. The only significant asymmetry is a small set of operations with semantics specific to their operand types.

Sometimes the orthogonality of operations and data types leads to a modicum of redundancy—for example, there is no semantic difference between the 'store signed longword' and 'store unsigned longword' instructions. There are relatively few such cases, however; and it could be argued that they offer an opportunity to make object programs somewhat self-explanatory.

Some of the decisions made in designing M may seem questionable. So much the better: that is grist to the mill of discussion, analysis, measurement, and revision. No provision has been made for operations that support legacy languages, such as packed decimal arithmetic for COBOL; nor are there any instructions operating in the SIMD mode on vector operands, such as are popular in commodity microprocessor designs. These gaps have been left, in part, to provide the motivation for students to design and implement new features themselves. Ample order code space has been reserved to allow extensions to be made cleanly and consistently. Some suggested extensions are described, and distinguished from the reference design by being set in red type.

M1 REGISTERS

Registers are used both as sources of operands and as destinations for results. These are two broad kinds of register—ARCHITECTURAL registers and IMPLEMENTATION registers. Architectural registers have the same functionality in all implementations of M. Implementation registers help to realize the architecture and vary from one implementation to another. In following algorithmic fragments, the name of a register or storage location denotes its contents as the source of a data flow, when appropriate, and its identity as the destination of a data flow, when that is appropriate.

M1.1 Implementation registers

In addition to the architectural registers, a realization of the architecture may include a number of IMPLEMENTATION REGISTERS, a selection of which may be displayed in diagnostic messages from an emulator. There are no explicit operations on implementation registers; they are accessed implicitly in the course of instruction execution. Those used in the current reference implementation include:

Name	Functionality
CIR	Current Instruction Register
OIA	OBEY Instruction Address
MAR	Memory Address Register
MBR	Memory Buffer Register

TABLE 1.1A: IMPLEMENTATION REGISTERS

M1.2 Architectural registers

There are 16 GENERAL REGISTERS, designated GR0…GRF; 16 FLOATING POINT REGISTERS, DR0…DRF; and 16 SPECIAL REGISTERS, SR0…SRF. Each process effectively has its own instance of each of these three register sets. When executing in kernel state, a further 16 KERNEL REGISTERS, designated KR0…KRF, become available. All of these registers are 64 bits wide. Unless otherwise stated, an operation updating a register affects the entire contents of the register. When the new value occupies less than 64 bits, it is extended before use to a 64-bit longword in an appropriate manner.

Each register bank has a conceptual bank number, which is used systematically to derive opcode variants for instruction classes that apply orthogonally to several different banks. This is 0 for the general registers, 1 for the special registers, 2 for the kernel registers, and 3 for the floating point registers.

M1.2.0 General registers

The GENERAL REGISTERS have a full complement of logical and integer arithmetic operations, and are specially equipped to hold operand addresses and to act as index registers. Four of the general registers have aliases, indicative of their particular functionality, as shown in Table 1.2.

#	Name	Functionality of GR#
C	EPR	EXTERNAL POINTER REGISTER, optionally used to address data areas outside the current stack frame
D	SPR	STACK POINTER REGISTER, pointing at all times to the first unused location in the stack
E	FPR	FRAME POINTER REGISTER, providing a base address for local variables (held in the current stack frame)
F	IAR	INSTRUCTION ADDRESS REGISTER, holding at all times the address of the following instruction; incremented (by 4) after fetching an instruction, but before executing it

TABLE 1.2A: GENERAL REGISTER FUNCTIONALITY

M1.2.1 Floating point registers

The FLOATING POINT REGISTERS provide for floating point expression evaluation. Floating point operations always involve the full precision of a floating point register. A single-precision (32-bit) floating point operand is widened to 64 bits before taking part in the operation. Floating point operands are in IEEE format.

If a floating point arithmetic operation has a result that is not representable in 64 bits then—depending on the floating point features of the host computer—it might either cause an Operand/Result Out Of Range (ORR) trap, or it might generate an IEEE NaN or Inf value in the result register. See §M11.

M1.2.2 Special registers

The SPECIAL REGISTERS serve a variety of purposes. They are accessible only by instructions that copy them to or from the general registers. SR4-SR7 are reserved for extensions to the M architecture.

#	Name	Register	Explanation
0	QMD	Quadruple length Multiply/Divide register	Set to the **rem** remainder of an integer division. Set to the lower half of a quadruple-word product. Taken as the lower half of a quadruple-word dividend, and set to the lower half of a quadruple-word quotient.
1	BTR	Boolean Test Register	Set to the Boolean result of a comparison or other test operation.
2	ENR	Exception Number Register	The number of the trap most recently caused by the current process.
3	TER	Trap Enable Register	A mask indicating the traps that are enabled for handoff.
8	OIS	Obeyed Instruction Successor	The address of the instruction that would have followed an OBEYed instruction, had it been executed in the course of normal control flow.
9	TVR	Trap Vector Register	The address of the trap routine address vector.
A	RWF	Read-Write Fence	The lowest address of a non-Read-Only location, typically set to the address of the start or the end of the program 'text' segment.
B	HLT	Heap LimiT	The limit address of the heap. See §M8.3.
C	SLT	Stack LimiT	The limit address of the stack. See §M8.3 and §M10.2.
D	SPC	Stack Pointer Copy	The value of the stack pointer saved by a CALL SPR instruction and used by a RETS instruction.
E	FPC	Frame Pointer Copy	The value of the frame pointer saved by a CALL FPR instruction and used by an ENTER instruction.
F	IAC	Instruction Address Copy	The value of the instruction address register saved by any call instruction and used by an ENTER, RETS or RETL instruction.

TABLE 1.2B: THE SPECIAL REGISTERS AND THEIR FUNCTIONALITY

M1.2.2 Kernel registers

The KERNEL REGISTERS provide for system monitoring, command, and control purposes. They are accessible only by instructions that copy them to or from the general registers. Access to KR0–KRB is illegal in user state and virtualized in virtual kernel state. KRC–KRF are readable in user state; they act somewhat as extensions to the special register set and their values have to be saved/restored on context switches. KR4-KR5 and KR8-KRB are reserved for extensions to the M architecture.

#	Name	Register	Explanation
0	ESF	Execution State Flags	The flags that enable and/or flag the various CPU states. See below.
1	AVR	Address Virtualization Register	The address of the virtual storage mapping data for the current process.
2	IDN	Interrupting Device Number	The device number of the external unit responsible for the currently actioned interrupt.
3	PDI	Pending Device Interrupts	Bit *i* is set iff external unit *i* is requesting an interrupt.
6	LAR	Linked Address Register	The address of a shared location set by a LDLNK instruction.
7	IRA	Interrupt Return Address	A copy of the value in IAR at the point of a taken external interrupt.
C	CCR	Calendar-Clock Register	The date and time in BCD format, updated every second.
D	NSR	Nanoseconds Register	The execution time of the current process, in nanoseconds, with an implementation-defined resolution.
E	ICR	Instruction Count Register	A count of the number of instructions that have completed execution in the current process.
F	TRA	Trap Return Address register	A copy of the value in IAR at the point of a taken trap.

TABLE 1.2C: THE KERNEL REGISTERS AND THEIR FUNCTIONALITY

The value obtained by reading any of the kernel registers not listed above is 0.

The ESF register is dedicated to CPU execution-state flags, including the bits that distinguish:

- **kernel state** (0) from **user state** (1): bit 0
- **real** state (0) from **virtualized** state (1): bit 1
- **non-OBEYed state** (0) from **OBEYed state** (1): bit 2; and
- **untrapped** state (0) from **trap-handling** state (1): bit 3.

Bit 6 is set by the LDLNK instruction, and cleared by STSKC and other events.

Bit 7 inhibits the taking of external device interrupts.

Bits 8 through 15 contain the CPU's interrupt priority level (presently limited to the range 0..15).

The remaining bits of the ESF register are RFE and should be zero.

The Calendar-Clock Register represents the current civil time as 16 decimal digits, each held in a single hexadecimal digit (Binary Coded Decimal, or BCD format). Expressed as *yyyy mm dd hh nn ss uw*, *yyyy* is the year number; *mm* is the month number within the year, counting from 1 for January; *hh*, *nn*, and *ss* are the time of day in hours, minutes and seconds; *u* is unused; and *w* is the week day, counting from 0 for Monday to 6 for Sunday. For example, at time of writing, it contains the hexadecimal value 2018080115055502, representing Wednesday, 2018-Aug-01 at 15:05:55.

The Nanoseconds Register is incremented regularly (by an implementation-defined amount) while a process is executing, and so provides a measure of its own consumption of CPU time. Successive reads of the Nanoseconds Register return unequal values, permitting their use as unique timestamps.

The Instruction Count Register is incremented by one each time a process completes the execution of an instruction in non-kernel state and thus provides a unique index of its execution. It is useful for measuring the computational complexity of code segments, and other software metrics.

In order to facilitate recursive virtualization, access to the kernel registers is limited if the CPU is not in real kernel state. In virtual kernel state, read accesses are 'censored' – that is to say, reads are masked so as not to divulge the virtualization; and writes always cause the Virtual Machine Monitor (VMM) trap to real kernel state, so that a virtual machine monitor can track the virtualized CPU state.

M2 Data and instruction formats

M2.1 Data formats

M supports signed (2's complement) and unsigned integers in 8-bit (BYTE), 16-bit (HALFWORD), 32-bit (WORD), and 64-bit (LONGWORD) representations. Floating point numbers occupy a word ('E format'), or, in double precision, a longword ('D format'); their representation is implementation defined.

The M convention is to use 'little-endian' bit numbering, the least significant bit of an item being numbered 0. In diagrams less-significant bits are shown to the right, and more-significant bits to the left. (Bits *i* through *j* of a quantity *Q* are notated thus: *Q* **bits** *i*:*j*; Bit *i* alone is notated thus: *Q* **bit** *i*.) The order of the bytes within a stored halfword, word or longword, the order of the halfwords within a stored word or longword, and the order of the words in a stored longword are consistently little-endian.

In the following, the notation `@`α denotes the storage location whose address is given by α.

M2.2 Instruction formats and operand specification

All instructions take up one word and are word-aligned (i.e. they begin on a word boundary, so that they always have an address that is an integral multiple of 4). All instructions have a 4-bit OPERATION GROUP field (G), in bits 4 through 7. There are three distinct instruction formats: the PRIMARY, SECONDARY and TERTIARY formats. Each format has a number of other fields that further specify the operation, but not all of these fields are used by every instruction. Any unused field is **r**eserved **f**or **e**xtensions (RFE) to the M architecture and is ignored. To ensure compatibility with future versions of M, all such fields must be coded as zero bits; the effect of non-zero bits is implementation defined, and may be indeterminate.

The PRIMARY FORMAT is made up of the following fields:

16 bits	4 bits	4 bits	4 bits	4 bits
N_p	X	R	G	T

where: R specifies a register operand; T extends the opcode bits; and the OPERAND PART of the instruction consists of X, the index register field, and N_p, a 16-bit constant. This is the format used for all branch instructions and most memory reference instructions.

The EFFECTIVE ADDRESS (`EA`) of a stored operand is given by $(n + x)$, where: x is the contents of `GR(X)`, considered as a 64-bit signed integer, **unless** X=0, in which case x=0 to provide a convenient base for ABSOLUTE addressing regardless of the contents of `GR(0)`; and n is the value of N_p, sign-extended to 64 bits.

The OPERAND VALUE in the IMMEDIATE operand mode is $(n + x)$, where: x is the contents of `GR(X)`, considered as a 64-bit signed integer, **unless** X=0, in which case x=0 to provide for CONSTANT immediate operands.

The additions performed in calculating $(n + x)$ wrap around on overflow and do not cause a process trap.

Although addresses in registers are 64 bits wide, and 64-bit address arithmetic is fully supported in the architecture, implementations may impose limits on the range of effective addresses. Future implementations may support sparse 64-bit virtual addressing; but present implementations, typically, provide only a few megabytes. (For the limits in current emulators, see §11.)

In the store-immediate instructions, R is taken to be a literal value, and not a general register designator.

The SECONDARY FORMAT has the following fields, where E extends the opcode; and N_s is a 12-bit constant:

12 bits	4 bits	4 bits	4 bits	4 bits	4 bits
N_s	E	X	R	G	T

Some less-used memory reference instructions take this format. The EFFECTIVE ADDRESS (`EA`) of a stored operand is given by $(n + x)$, where n is the value of N_s, sign-extended to 64 bits. Immediate operands are derived analogously. In the floating point store-immediate instructions, R is taken to be an index into a table of constant values, and not a floating point register designator.

A possible extension of the architecture is to use the low order bits of halfword, word and longword addresses of aligned store accesses, normally 0, to indicate scaling of x, computing the effective address as $(n + x<<s)$, where s is 0 or 1 for a halfword operand, in 0..3 for a word operand, and in 0..7 for a doubleword.

The TERTIARY FORMAT has the following fields, where Y specifies a register; and N_t is an 8-bit constant:

8 bits	4 bits	4 bits	4 bits	4 bits	4 bits	4 bits
N_t	Y	E	X	R	G	T

Some instructions use the immediate value $(n + x)$, where n is the value of N_t, sign-extended to 64 bits.

M3 PROCESS TRAPS AND INTERRUPTS

The occurrence of any of several errors causes a PROCESS TRAP, an unintended interruption of normal control flow. If an instruction causes a process trap, the value left in any destination operand (whether RAM location or register) is implementation defined, and may be indeterminate.

The process traps currently implemented by the architecture are shown in Table 3.1A.

Read-Only Violation	ROV	an attempt to write into a read-only location	1
Operand Alignment Violation	OAV	an attempt to access an incorrectly aligned operand	2
Invalid Physical Address	IPA	An attempt to access a non-existent location	3
Stack Limit Violation	SLV	a stack instruction that sets SPR or FPR to a value less than that held in SLT; or a TRIM instruction with invalid operand	4
User-state Privilege Violation	UPV	an attempt, in user state, to execute an instruction that may be executed only in the kernel state	5
Reserved Instruction Format	RIF	executing a bit pattern that has been set aside as not representing a valid instruction, whether permanently or for future extension (any bit pattern not otherwise accounted for can be assumed to be in this category)	6
Instruction Not Yet Implemented	NYI	an instruction that is defined in the M architecture, but not yet implemented	7
Operand/Result out of Range	ORR	an instruction computing a result that overflows the representable range for the result; or having an operand outside the domain of the operation	8
Recursive OBEY Attempt	ROA	an OBEY instruction having an OBEY instruction as its operand	9
Address Range Error	ARE	TRAPA detected an erroneous **a**ddress, or other invalid address detection	A
Signed Range Error	SRE	TRAPB detected a signed integer out of range	B
Unsigned Range Error	URE	TRAPP detected an unsigned integer out of range	C
User-Specified Trap	UST	A low-numbered trap at the disposal of the user for use in explicitly-designated debugging traps	D
True Value Trap	TVT	A low-numbered trap at the disposal of the user for use in explicitly-designated debugging traps	E
False Value Trap	FVT	A low-numbered trap at the disposal of the user for use in explicitly-designated debugging traps	F
Virtual Machine Monitor	VMM	an instruction attempting an operation that needs virtualisation in the present CPU state	10_{16}
User Break-In	UBI	response to the OS being requested by the user (e.g. using ^C) to interrupt execution	63
Trap	TLA	Causation	Code

TABLE 3.1A: HARDWARE-DEFINED PROCESS TRAPS

Several dedicated special registers are used in the handling of traps. These are: the Exception Number Register, ENR; the Execution Status Flags Register, ESF; the Trap Vector Register, TVR; and the Trap Return Address, TRA. When a process trap occurs, its code number is placed in the Exception Number Register, ENR. In the event of errors discovered by the Operating System, or the emulator, or User-Specified Traps diagnosed by the program itself using the TRAPD, TRAPT or TRAPF orders, ENR is set to a diagnostic code rather than the trap number.

Emulator-detected error codes have E in the most significant hexadecimal digit of the most significant byte, and Operating System error codes have F in that position. The universally-defined emulator error codes are shown in Table 3.1B.

Error	Code
run stopped by user	E0_00_00_00_00_00_00_01
run terminated on reaching time limit (probable infinite loop)	E0_00_00_00_00_00_00_02
self-detected **e**rror in emulator	E0_00_00_00_00_00_00_0**E**
unanticipated **f**ailure of emulator	E0_00_00_00_00_00_00_0**F**

TABLE 3.1B: EMULATOR ERROR CODES

A trap may be handled either by the process that caused it, or by the Operating System, according to the options specified in the Trap Enable Register, TER. If TER **bit** i = 0, the trap with code number i is dealt with by the Operating System; if TER **bit** i = 1, trap i is HANDED OFF to the process.

When a process is already dealing with a handed-off trap, a further trap is dealt with by the Operating System. It is not possible to enable for hand-off a trap with number $i > 63$; if such a trap is caused, it is always dealt with by the Operating System.

A trap is handed-off to a process by:

- saving IAR in the Trap Return Address Register, TRA
- writing the trap's code number into ENR, and setting a state bit in ESF, to indicate that the process is handling a trap; in the case of trap D, the operand of the trap order is placed in ENR
- setting IAR to the address contained in the location whose address is given by the Trap Vector Register, TVR, and the trap's code number, i, thus: `IAR := @(TVR + 8*i).`

A process returns from a trap to the interrupted instruction by means of the Return from Trap (RETT) instruction, which:

- clears the ENR
- clears the trap-handling state bit in ESF
- jumps to the instruction at the address given by TRA**.**

All traps dealt with by the Operating System select a Trap Service Routine (TSR), using the trap number, via an entry in a table starting at an implementation dependent fixed address.

M3.1 The Operand/Result Out Of Range trap

The Operand/Result Out Of Range (ORR) trap will be caused:

- By a signed integer arithmetic operation having a result that is not representable in 64 bits.
- By division by zero.
- By a store instruction with partword or word operand. The trap is caused by these instructions if the value of the entire source is out of the range of the destination. For STRSB it must lie in the range [-128, +127]; for STRUB, in the range [0, +255]; for STRSH and SUSH, in the range [-32768, +32767]; for STRUH and SUUH, in the range [0, +65535]; for STRSW and SUSW, in the range $[-2^{31}, +2^{31}-1]$; and for STRUW and SUUW, in the range $[0, +2^{32}-1]$.
- By a STRE instruction if the content of the source register is not representable as a 32-bit floating point operand.
- By a CONSBR, CONSHR, CONSWR, CONUBR, CONUHR, CONUWR, or CONDR instruction if the content of the source register is not representable in the destination operand type.
- By a shift instruction, if the shift amount is negative, or greater than the operand size.
- By a trap instruction, if the operand is greater than $D_{16} \times 2^{60}$, considered as an unsigned number.
- By a floating point arithmetic operation having a result that is not representable in 64 bits—if the floating point feature of the host computer behaves in this manner, rather than generating an IEEE NaN or Inf value in the result register. See §M11.

M3.2 External interrupt and process trap mechanisms

External device 0 is the SYSTEM MONITOR UNIT (SMU). It notifies the CPU of failure conditions and I/O device interrupt requests. The latter are indicated by setting the corresponding bit in the PDI register, and effected when the priority of the request exceeds the current operating priority of the CPU **and** interrupts are not currently inhibited. Note that ESF **bit** 7 inhibits only the **taking** of any device interrupts. The **generation** of interrupt requests is enabled or disabled on a device-by device basis, using the DOIO instruction.

When an interrupt takes place, the number of the interrupting device is written into the INTERRUPTING DEVICE NUMBER register, IDN; the CPU kernel state bit and interrupt priority are set in the ESF register; and the CPU jumps to the First-Level Interrupt Handler (FLIH) at an implementation-defined address. Further action is vectored by the FLIH to an appropriate DEVICE SERVICE ROUTINE (DSR) using the contents of IDN.

The mapping from device to device-number is configuration dependent and may even be set dynamically.

The FLIH can allow higher-priority interrupts to take place by clearing ESF **bit** 7, once it has stabilized the machine state and established re-entrancy.

A vectored interrupt handler can schedule deferred processing of a response by requesting an Asynchronous Service Routine (ASR) trap, at priority level 4.

Process traps are treated similarly: the kernel and the trap-handling state bits are set in the ESF register; and the CPU jumps to the First-Level Trap Handler (FLTH) at an implementation-defined address different from that used by external interrupts. Further action is vectored by the FLTH to an appropriate TRAP SERVICE ROUTINE (TSR) using the contents of ENR.

There are 16 interrupt priority levels (where a higher level number means higher CPU priority):

- level F is for SMU-originated interrupts – **f**ailures, mostly – e.g. power fail, bad memory, etc.
- level E is for traps (other than interrupts) from real kernel state – **e**rrors in, or detected by, the kernel.
- levels 8 through D are for I/O **d**evice interrupts relayed by the SMU.
- level 4 is for activating Asynchronous Service Routines.
- level 0 is for traps (other than interrupts) from states other than real kernel state.

Normal (uninterrupted) processing also runs at level 0.

Levels 1 .. 3, and 5 .. 7, are reserved for extensions of the architecture.

The mapping from device to priority level is configuration dependent and may be dynamically dependent on the reason for the interrupt.

Each interrupt priority level has its own sets of general and special registers. Switching to level *n* automatically activates GR set *n* and SR set *n*, so that no delay in interrupt response is incurred for swapping register contents. (Interrupts seldom need access to the floating point registers.) System calls, extracodes, unhandled process traps, and so on, use the same set of registers as the code that was interrupted. This allows their trap handler easy access to parameters in the registers of the interrupted process.

To be augmented …

Trap enable bits …

M4 The groups 0-9 general-register memory reference ALU instructions

These primary-format instructions set GR(R) to the result of an ALU operation that takes as inputs GR(R) and a second operand. The latter is either an IMMEDIATE operand, held in the instruction itself, or a STORED operand, held in memory. The type of the operand, and its mode (immediate/stored) are determined by the T field. The ALU operation to be carried out is determined by the G field. A variant of this scheme is the STR (G=0) operation, which copies GR(R) into RAM, and does not admit an immediate operand. The encoding of G and T has been chosen so that small integers represent invalid instructions, thus increasing the likelihood that a wild jump into data will be quickly trapped.

M4.1 Operand addressing, modes and types

Operands are 1 (**byte**), 2 (**halfword**), 4 (**word**), or 8 (**longword**) bytes in length, as determined by T. The addressing unit is the byte. A stored operand must have an address that is an integral multiple of its length. Each program runs in its own 64-bit linear ADDRESS SPACE, of which only a subset may be implemented.

In the STORED OPERAND mode the EFFECTIVE ADDRESS (EA) of an operand is given by the formula ($n + x$); the OPERAND VALUE (V) (for G≠0) is @EA. Addressing with X=0 is ABSOLUTE; with X=IAR it is PROGRAM-RELATIVE; with X=FPR it is LOCAL; with X=SPR it is STACK-RELATIVE; and with X=EPR it is NON-LOCAL Thus the baroque prolixity of special addressing modes in commodity architectures is subsumed in M by the orthogonal combination of a small set of primitives.

In the IMMEDIATE OPERAND mode the operand value is determined by the operand size as shown in Table 4.1. An operand part with N=0 and X≠0 provides a simple means of using a general register as an operand in an instruction with a general register destination.

Immediate Operand , V	Operand Type
($n + x$) **bits** 0:7	**b**yte
($n + x$) **bits** 0:15	**h**alfword(2 bytes)
($n + x$) **bits** 0:31	**w**ord (4 bytes)
($n + x$)	**l**ongword (8 bytes)

Table 4.1: Immediate Operand Semantics

The value V may be further transformed to produce the final operand, in a manner depending on the particular instruction using the operand. For example, an unsigned instruction extends V to yield a longword by zero filling the most significant 56, 48 or 32 bits; and a signed instruction obtains a longword by propagating the sign of V (bit 7, 15 or 31) into the most significant 56, 48 or 32 bits.

T determines the operand type and mode as shown in Table 4.3, where each entry displays a value of T and the mnemonic operation suffix for the corresponding type and mode.

M4.2 Notational conventions

In the following tables of instructions in groups 0-9:

- **abs**, **mod** and **rem** are all defined as in Ada 2012.
- **or** is the bit-wise logical *or* operator, **xor** is logical *exclusive-or*, **and** is logical *and*, **not** is logical *not*.
- T(V) denotes the value of the operand, V, prepared as a 64-bit quantity according to the type specified by T, as described in §4.1.

M4.3 Orthogonal semantics

The instructions in Tables 4.4A-B have semantics that are an orthogonal combination of the operation (given by G) and the operand type (given by T). Table 4.3A gives the semantics of the operand types.

Byte	**H**alfword	**W**ord	**L**ongword	
0—SBI	1—SHI	2—SWI	3—SLI	**S**igned **I**mmediate
4—UBI	5—UHI	6—UWI	7—ULI	**U**nsigned **I**mmediate
8—SB	9—SH	A—SW	B—SL	**S**igned (stored)
C—UB	D—UH	E—UW	F—UL	**U**nsigned (stored)

Table 4.3A: Operand Type and Mode

Table 4.3B sets out the semantics of the operations; the notation T(V) denotes the operand value V, processed as required by the operand type T.

0	**store signed** (STRS) @EA := GR(R)	**store unsigned** (STRU) @EA := GR(R)
1	**load signed** (LODS) GR(R) := T(V)	**load unsigned** (LODU) GR(R) := T(V)
2	**add signed** (ADDS) GR(R) := GR(R) + T(V)	**add unsigned** (ADDU) GR(R) := GR(R) + T(V)
3	**subtract signed** (SUBS) GR(R) := GR(R) - T(V)	**subtract unsigned** (SUBU) GR(R) := GR(R) - T(V)
4	**multiply signed** (MULS) GR(R) := GR(R) * T(V)	**multiply unsigned** (MULU) GR(R) := GR(R) * T(V)
5	**divide signed** (DIVS) QMD := GR(R) **rem** T(V) GR(R) := GR(R) / T(V)	**divide unsigned** (DIVU) QMD := GR(R) **rem** T(V) GR(R) := GR(R) / T(V)
6	**multiply quadword signed** (MLQS) GR(R):QMD := GR(R) * T(V)	**logical inclusive or** (IORU) GR(R) := GR(R) **or** T(V)
7	**divide quadword signed** (DVQS) QMD := (GR(R):QMD / T(V)) **bits** 0:63 GR(R) := (GR(R):QMD / T(V)) **bits** 64:127	**logical exclusive or** (XORU) GR(R) := GR(R) **xor** T(V)
8	**negate (two's complement)** (NEGS) GR(R) := **-** T(V)	**not (one's complement)** (NOTU) GR(R) := **not** T(V)
9	**modulus** (MODS) GR(R) := GR(R) **mod** T(V)	**logical and** (ANDU) GR(R) := GR(R) **and** T(V)
G	signed operands: T ∈ {0..3, 8..B}	unsigned operands: T ∈ {4..7, C..F}

TABLE 4.3B: OPERATIONS AND THE STEMS OF THEIR MNEMONIC OPCODES

The MLQSL, MLQSLI, DVQSL and DVQSLI instructions work with 128-bit quantities.

MLQSL and MLQSLI produce the 128-bit product of two 64-bit values, placing the less significant 64 bits in QMD.

DVQSL and DVQSLI divide the 128-bit dividend in QMD and GR(R)—the less significant 64 bits being in QMD—by the 64-bit T(V) operand, giving a 128-bit quotient in GR(R) and QMD.

With the STR{SU}{BHWL}I instructions, the immediate operand is the value of the R field, **not** the value $(n + x)$, and the value stored is **not** GR(R). In the case of the STRU{BHWL}I instructions, the exact value of R is used, so the range that can be stored is 0..15. In the case of the STRS{BHWL}I instructions, R is sign-extended to 64 bits, so the range that can be stored is -8..+7. These instructions are written in A with the literal operand where the GR(R) field would otherwise be placed. Executing a store-immediate instruction that has N=0 and X=0 is illegal and causes the Reserved Instruction Format trap. This restriction means that all words with value less than 4096 are illegal when considered as instructions, so that wild jumps into data are likely to fail quickly.

0	STRSBI	STRSHI	STRSWI	STRSLI	STRSB	STRSH	STRSW	STRSL
1	LODSBI	LODSHI	LODSWI	LODSLI	LODSB	LODSH	LODSW	LODSL
2	ADDSBI	ADDSHI	ADDSWI	ADDSLI	ADDSB	ADDSH	ADDSW	ADDSL
3	SUBSBI	SUBSHI	SUBSWI	SUBSLI	SUBSB	SUBSH	SUBSW	SUBSL
4	MULSBI	MULSHI	MULSWI	MULSLI	MULSB	MULSH	MULSW	MULSL
5	DIVSBI	DIVSHI	DIVSWI	DIVSLI	DIVSB	DIVSH	DIVSW	DIVSL
6	MLQSBI	MLQSHI	MLQSWI	MLQSLI	MLQSB	MLQSH	MLQSW	MLQSL
7	DVQSBI	DVQSHI	DVQSWI	DVQSLI	DVQSB	DVQSH	DVQSW	DVQSL
8	NEGSBI	NEGSHI	NEGSWI	NEGSLI	NEGSB	NEGSH	NEGSW	NEGSL
9	MODSBI	MODSHI	MODSWI	MODSLI	MODSB	MODSH	MODSW	MODSL
G	T=0	T=1	T=2	T=3	T=8	T=9	T=A	T=B

TABLE 4.3C: THE SIGNED INTEGER INSTRUCTIONS

0	STRUBI	STRUHI	STRUWI	STRULI	STRUB	STRUH	STRUW	STRUL
1	LODUBI	LODUHI	LODUWI	LODULI	LODUB	LODUH	LODUW	LODUL
2	ADDUBI	ADDUHI	ADDUWI	ADDULI	ADDUB	ADDUH	ADDUW	ADDUL
3	SUBUBI	SUBUHI	SUBUWI	SUBULI	SUBUB	SUBUH	SUBUW	SUBUL
4	MULUBI	MULUHI	MULUWI	MULULI	MULUB	MULUH	MULUW	MULUL
5	DIVUBI	DIVUHI	DIVUWI	DIVULI	DIVUB	DIVUH	DIVUW	DIVUL
6	IORUBI	IORUHI	IORUWI	IORULI	IORUB	IORUH	IORUW	IORUL
7	XORUBI	XORUHI	XORUWI	XORULI	XORUB	XORUH	XORUW	XORUL
8	NOTUBI	NOTUHI	NOTUWI	NOTULI	NOTUB	NOTUH	NOTUW	NOTUL
9	ANDUBI	ANDUHI	ANDUWI	ANDULI	ANDUB	ANDUH	ANDUW	ANDUL
G	T=4	T=5	T=6	T=7	T=C	T=D	T=E	T=F

TABLE 4.3D: THE UNSIGNED INTEGER INSTRUCTIONS

M5 The group A Addressing and Alignment instructions

Group A might be used to contain primary-format instructions useful for setting up addresses in index registers (INSMWI, INSMW and EXTMW); and for accessing longword, word or halfword stored operands that do not have natural address alignment (e.g. within arrays of packed records).

The effective address used by the INSMW and EXTMW orders must be word-aligned.

Loading or storing an unaligned operand may cross a longword boundary, requiring access to two successive longwords. If the operand is wholly contained in one longword, only that one longword is accessed.

The halfword- and word- store instructions check the contents of GR(R) for a value out of the range of the destination, in the same way as the aligned-operand store orders. As with the latter, there is no difference between the LUSL and LUUL instructions or between the SUSL and SUUL instructions in this respect, as it is always possible to store the contents of a register into a longword.

The instruction is determined by T, see Table 5.1A.

0	INSMWI	**ins**ert into **m**ore significant **w**ord of register **i**mmediate	GR(R) **bits** 32:63 := (*n* + *x*) **bits** 0:31; GR(R) **bits** 0:31 are unchanged
1	LUSH	**l**oad **u**naligned **s**igned **h**alfword	GR(R) **bits** 0:15 := @(EA..EA+1); GR(R) **bits** 16:63 := GR(R) **bit** 15
2	LUSW	**l**oad **u**naligned **s**igned **w**ord	GR(R) **bits** 0:31 := @(EA..EA+3); GR(R) **bits** 32:63 := GR(R) **bit** 31
3	LUSL	**l**oad **u**naligned **s**igned **l**ongword	GR(R) **bits** 0:63 := @(EA..EA+7)
4	RFE		
5	LUUH	**l**oad **u**naligned **un**signed **h**alfword	GR(R) **bits** 0:15 := @(EA..EA+1); GR(R) **bits** 16:63 := 0
6	LUUW	**l**oad **u**naligned **un**signed **w**ord	GR(R) **bits** 0:31 := @(EA..EA+3); GR(R) **bits** 32:63 := 0
7	LUUL	**l**oad **u**naligned **un**signed **l**ongword	GR(R) **bits** 0:63 := @(EA..EA+7)
8	INSMW	**ins**ert into **m**ore significant **w**ord of register	GR(R) **bits** 32:63 := @EA **bits** 0:31; GR(R) **bits** 0:31 are unchanged
9	SUSH	**s**tore **u**naligned **s**igned **h**alfword	@(EA..EA+1) := GR(R)
A	SUSW	**s**tore **u**naligned **s**igned **w**ord	@(EA..EA+3) := GR(R)
B	SUSL	**s**tore **u**naligned **s**igned **l**ongword	@(EA..EA+7) := GR(R)
C	EXTMW	**ext**ract **m**ore significant **w**ord from register (aligned)	@(EA..EA+3) := GR(R) **bits** 32:63
D	SUUH	**s**tore **u**naligned **un**signed **h**alfword	@(EA..EA+1) := GR(R)
E	SUUW	**s**tore **u**naligned **un**signed **w**ord	@(EA..EA+3) := GR(R)
F	SUUL	**s**tore **u**naligned **un**signed **l**ongword	@(EA..EA+7) := GR(R)
E	Mnemonic	Operation	Effect

Table 5.1A: Unaligned/Extract/Insert Operations

M6 THE GROUP B BRANCH INSTRUCTIONS

Instructions of group B, the BRANCH GROUP, are encoded in the primary format, with T determining the BRANCH TYPE, as shown in Table 6.1A. The new address placed in the instruction address register (IAR) to effect the branch is the 64-bit value $(n + x)$, computed as usual, for T ∈ {0, 2..7, 8, A..F}; and is the contents of the 64-bit word @$(n + x)$, for T ∈ {1, 9}.

T	Mnemonic	Condition	T	Mnemonic	Condition
0	BASRA	always, saving IAR in GR(R)	8	CALL	always, saving return address(es)
1	BASVIA	always, saving IAR in GR(R)	9	CALVIA	always, saving return address(es)
2	BEQZ	GR(R) = 0	A	BEQZD	DR(R) = 0
3	BLTZ	GR(R) < 0	B	BLTZD	DR(R) < 0
4	BLEZ	GR(R) ≤ 0	C	BLEZD	DR(R) ≤ 0
5	BGTZ	GR(R) > 0	D	BGTZD	DR(R) > 0
6	BGEZ	GR(R) ≥ 0	E	BGEZD	DR(R) ≥ 0
7	BNEZ	GR(R) ≠ 0	F	BNEZD	DR(R) ≠ 0

TABLE 6.1A: THE BRANCH INSTRUCTIONS

Specifying X=0 in these instructions gives ABSOLUTE branches.

Specifying X=IAR gives RELATIVE branches, allowing for POSITION-INDEPENDENT CODE (PIC).

The BASRA (**b**ranch **a**nd **s**ave **r**eturn **a**ddress) instruction saves IAR in GR(R), before setting IAR to the operand value. When R=F (IAR), BASRA effects a simple unconditional jump.

BASRA GR(R) *subroutine* alone is sufficient to implement the simplest possible call-and-return protocol, return being effected by BASRA IAR 0[GR(R)], with GR(R) the same register as in the call.

The CALL instruction provides the additional mechanisms needed by more general calling protocols, including recursion. CALL saves IAR in IAC, and GR(R) in SR(R), before setting IAR to the operand value. So, CALL SPR saves SPR in SPC; and CALL FPR saves FPR in FPC.

The BASVIA and CALVIA instructions bear the same relation to BASRA and CALL (respectively) as the stored operand mode does to the immediate operand mode; namely, that their branch address is the contents of the 64-bit word whose address is given by $(n + x)$. These **indirect** branch instructions are useful in implementing the 'dispatching' or 'method' calls of Object-Oriented programming, as well as for the less trendy 'case' statement.

BEQZ and BNEZ are often used to act on a Boolean computed by one of the test group of instruction (see §M7.1 and §M7.2). In those roles the A programmer may helpfully employ instead their assembly language synonyms: BFALSE and BTRUE, respectively.

Note the absence of a plain unconditional branch from this group. The GOTO and GOVIA assembler mnemonics provide direct and indirect unconditional branches, respectively. These are assembly language synonyms for BASRA IAR and BASVIA IAR respectively, exploiting the fact that IAR is a first-class member of the general register set.

Similarly the GOSUB and GOBACK assembler mnemonics provide convenient abbreviations for BASRA GR(R) and BASRA IAR respectively.

Bits 0 and 1 of N_p in these instructions are redundant, normally 0, so there are possible extensions that would exploit them for greater functionality. Here are some examples:

• for BASRA, BASVIA, CALL and CALVIA: to make indexed jumps easier, with bits 0 and 1 of $N_p = 10$: take the effective address to be ($n^* + x<<2$), where n^* is n with the least significant 2 bits set to 0 and <<2 is a signed (arithmetic) left shift of length 2

• for T ∈ {2..7}, with bits 0 and 1 of $N_p = 01$:

2	INCEQZ	GR(R) := GR(R)+1; GR(R) = 0
3	INCLTZ	GR(R) := GR(R)+1; GR(R) < 0
4	INCLEZ	GR(R) := GR(R)+1; GR(R) ≤ 0
5	INCGTZ	GR(R) := GR(R)+1; GR(R) > 0
6	INCGEZ	GR(R) := GR(R)+1; GR(R) ≥ 0
7	INCNEZ	GR(R) := GR(R)+1; GR(R) ≠ 0
T	Mnemonic	Condition

TABLE 6.1B: THE INCREMENT-AND-TEST BRANCH INSTRUCTIONS

• for T ∈ {2..7}, with bits 0 and 1 of N_p =11:

2	DECEQZ	GR(R) := GR(R)-1; GR(R) = 0
3	DECLTZ	GR(R) := GR(R)-1; GR(R) < 0
4	DECLEZ	GR(R) := GR(R)-1; GR(R) ≤ 0
5	DECGTZ	GR(R) := GR(R)-1; GR(R) > 0
6	DECGEZ	GR(R) := GR(R)-1; GR(R) ≥ 0
7	DECNEZ	GR(R) := GR(R)-1; GR(R) ≠ 0
T	Mnemonic	Condition

TABLE 6.1C: THE DECREMENT-AND-TEST BRANCH INSTRUCTIONS

• for T ∈ {A..F}, with bits 0 and 1 of $N_p = 01$:

A	BEQNZ	DR(R) = -0.0
B	BNINF	DR(R) = *-Inf*
C	BNAND	DR(R) is a *NaN*
D	BPINF	DR(R) = *+Inf*
E	BNUMD	DR(R) is not a *NaN*
F	BNENZ	DR(R) ≠ -0.0
T	Mnemonic	Condition

TABLE 6.1D: THE EXCEPTIONAL FLOATING-POINT NUMBER BRANCH INSTRUCTIONS

M7 THE GROUP C COMPUTE, CONVERT, COMPARE AND COPY INSTRUCTIONS

Group C contains subgroups providing REGISTER-TO-REGISTER OPERATIONS. These tertiary instructions are first determined by T: for T ∈ [0, 7] they are general-register-to-register ALU operations; for T=8 they are floating point register-to-register operations; for T=9 they are set-to-set register copying operations; and the others are RFE.

M7.1 General-register-to-register computation subgroup

Unless otherwise stated, these instructions involve an expression of the form `GR(Y)` *op* T(V); where *op* is given by E, and T(V) is an immediate value derived from (*n* + *x*), according to the operand type, in the manner given in §4.1. The monadic operations take only a T(V) operand.

The instructions (E=0 : T ∈ {0, 1, 2, 4, 5, 6}), namely `CONSBR`, `CONSHR`, `CONUBR`, `CONUHR`, `CONSWR` and `CONUWR`, are provided to **con**vert longwords to partwords or words, checking that the value of the source operand lies in the partword or word range. The result overwrites all 64 bits of the destination register. The ORR trap is caused if the range check fails. These instructions are identity transformations so long as the values of their source operands lie in the range of the partword. When the source and destination registers are the same, they offer a convenient means of validating a sub-expression, leaving it unchanged in both value and position.

The instructions (G=C : E ∈ [A, F] : T∈ [0, 7]), perform a comparison: `GR(Y)` *rel* V, where *rel* is one of "=", "<", "≤", ">", "≥", "≠", as determined by E. The Boolean result of the comparison, represented by 0 for False and 1 for True, is left in both `GR(R)` and `BTR`. The result in `GR(R)` is convenient for testing with a branch order; the copy in `BTR` enables the *select* orders (see §M7.5).

0	`CONSBR`
	trap unless
	V ∈ [-128, +127];
	`GR(R) := V`
		`CONSHR`
		trap unless
		V ∈ [-32768, +32767];
		`GR(R) := V`
			`CONSWR`
			trap unless
			V ∈ [-2^{31}, $+2^{31}-1$];
			`GR(R) := V`
				RFE

1	`LODSBR`	`LODSHR`	`LODSWR`	`LODSLR`
2	`ADDSBR`	`ADDSHR`	`ADDSWR`	`ADDSLR`
3	`SUBSBR`	`SUBSHR`	`SUBSWR`	`SUBSLR`
4	`MULSBR`	`MULSHR`	`MULSWR`	`MULSLR`
5	`DIVSBR`	`DIVSHR`	`DIVSWR`	`DIVSLR`
6	`MLQSBR`	`MLQSHR`	`MLQSWR`	`MLQSLR`
7	`DVQSBR`	`DVQSHR`	`DVQSWR`	`DVQSLR`
8	`NEGSBR`	`NEGSHR`	`NEGSWR`	`NEGSLR`
9	`MODSBR`	`MODSHR`	`MODSWR`	`MODSLR`
A	`TEQSBR`	`TEQSHR`	`TEQSWR`	`TEQSLR`
B	`TLTSBR`	`TLTSHR`	`TLTSWR`	`TLTSLR`
C	`TLESBR`	`TLESHR`	`TLESWR`	`TLESLR`
D	`TGTSBR`	`TGTSHR`	`TGTSWR`	`TGTSLR`
E	`TGESBR`	`TGESHR`	`TGESWR`	`TGESLR`
F	`TNESBR`	`TNESHR`	`TNESWR`	`TNESLR`
E	T= 0	T= 1	T= 2	T= 3

TABLE 7.1A: THE SIGNED INTEGER REGISTER-REGISTER INSTRUCTIONS

0	`CONUBR`
	trap unless
	V ∈ [0, 255];
	`GR(R) := V`
		`CONUHR`
		trap unless
		V ∈ [0, 65535];
		`GR(R) := V`
			`CONUWR`
			trap unless
			V ∈ [0, +2^{32}-1];
			`GR(R) := V`
				RFE

1	`LODUBR`	`LODUHR`	`LODUWR`	`LODULR`
2	`ADDUBR`	`ADDUHR`	`ADDUWR`	`ADDULR`
3	`SUBUBR`	`SUBUHR`	`SUBUWR`	`SUBULR`
4	`MULUBR`	`MULUHR`	`MULUWR`	`MULULR`
5	`DIVUBR`	`DIVUHR`	`DIVUWR`	`DIVULR`
6	`IORUBR`	`IORUHR`	`IORUWR`	`IORULR`
7	`XORUBR`	`XORUHR`	`XORUWR`	`XORULR`
8	`NOTUBR`	`NOTUHR`	`NOTUWR`	`NOTULR`
9	`ANDUBR`	`ANDUHR`	`ANDUWR`	`ANDULR`
A	`TEQUBR`	`TEQUHR`	`TEQUWR`	`TEQULR`
B	`TLTUBR`	`TLTUHR`	`TLTUWR`	`TLTULR`
C	`TLEUBR`	`TLEUHR`	`TLEUWR`	`TLEULR`
D	`TGTUBR`	`TGTUHR`	`TGTUWR`	`TGTULR`
E	`TGEUBR`	`TGEUHR`	`TGEUWR`	`TGEULR`
F	`TNEUBR`	`TNEUHR`	`TNEUWR`	`TNEULR`
E	T= 4	T= 5	T= 6	T= 7

TABLE 7.1B: THE UNSIGNED INTEGER REGISTER-REGISTER INSTRUCTIONS

M7.2 Register-to-register floating point computation subgroup

Register-to-register floating point operations involve a double-precision expression `DR(Y)` *op* `DR(X)`. There are no register-to-register single-precision operations, as the floating point registers always contain double-precision (D format) values.

On some hosts floating point arithmetic errors cause an ORR trap; but on others they may instead generate a NaN (not-a-number) value or an Inf (infinity) value. To avoid imposing unacceptable overheads, the behaviour of M's floating point operations in response to such errors is implementation defined.

The instructions (E ∈ [A, F] : T=8) generate a Boolean result from the comparison `DR(Y)` *rel* `DR(X)`, where *rel* is one of "=", "<", "≤", ">", "≥", "≠", as determined by E. The Boolean result of the comparison, represented by 0 for False and 1 for True, is left in both `GR(R)` and `BTR`. The result in `GR(R)` is conventient for testing with a branch order; the copy in `BTR` enables the *select* orders (see §M7.5).

In Table 7.2A:

- **round** yields its operand rounded to the nearest integer, in 64-bit floating point
- **floor** yields its operand adjusted to the nearest integer of lesser or equal value, in 64-bit floating point
- **ceiling** yields its operand adjusted to the nearest integer of greater or equal value, in 64-bit floating point
- **long** yields the 64-bit (signed longword) integer nearest to a 64-bit floating point value.

0	CONDR **trap unless** DR(X) ∈ [-2^{63}, +2^{63}-1]; GR(R) := **long** DR(X)
1	LODDR DR(R) := DR(X)
2	ADDDR DR(R) := DR(Y) + DR(X)
3	SUBDR DR(R) := DR(Y) – DR(X)
4	MULDR DR(R) := DR(Y) * DR(X)
5	DIVDR DR(R) := DR(Y) / DR(X)
6	RNDDR DR(R) := **round** DR(X)
7	FLRDR DR(R) := **floor** DR(X)
8	NEGDR DR(R) := - DR(X)
9	CLGDR DR(R) := **ceiling** DR(X)
A	TEQDR BTR,GR(R) := DR(Y) = DR(X)
B	TLTDR BTR,GR(R) := DR(Y) < DR(X)
C	TLEDR BTR,GR(R) := DR(Y) ≤ DR(X)
D	TGTDR BTR,GR(R) := DR(Y) > DR(X)
E	TGEDR BTR,GR(R) := DR(Y) ≥ DR(X)
F	TNEDR BTR,GR(R) := DR(Y) ≠ DR(X)
E	T= 8

TABLE 7.2A: THE FLOATING POINT REGISTER-TO-REGISTER INSTRUCTIONS

M7.3 Set-to-set register copying subgroup

These operations are in the subgroup with T=9. They provide for verbatim copying of bit patterns between registers in different sets and between different registers in the same set.

E	Operation
0	`CPYGG` `GR(R) := GR(X)`
1	`CPYSG` `GR(R) := SR(X)`
2	`CPYKG` `GR(R) := KR(X)`
3	`CPYDG` `GR(R) := DR(X)`
4	`CPYGS` `SR(R) := GR(X)`
5	`CPYSS` `SR(R) := SR(X)`
8	`CPYGK` `KR(R) := GR(X)`
A	`CPYKK` `KR(R) := KR(X)`
C	`CPYGD` `DR(R) := GR(X)`
F	`CPYDD` `DR(R) := DR(X)`

TABLE 7.3A: THE SET-TO-SET REGISTER COPYING INSTRUCTIONS

`CPYKK` and `CPYGK` are legal only in kernel state and cause the UPV trap in user state.
`CPYKG` is legal in kernel state; it is illegal in user state, causing the UPV trap, **unless** `X` is in the range 8..F.

M7.4 Push and pop instructions

The tertiary instructions G=C; T=A; provide stack push and pop operations for all register sets and stacks oriented in the 'natural' direction, i.e. downward. These are useful in implementing lightweight subroutine mechanisms (*inter alia*). It is also useful to see them as autodecrement/autoincrement 'register indirect' fetch and store instructions. They are as shown in Table 7.4A.

E	Operation
0	`PUSHDNB` **pushdown** `GR(R)` **bits** 56:63, `GR(X)`
1	`PUSHDNH` **pushdown** `GR(R)` **bits** 48:63, `GR(X)`
2	`PUSHDNW` **pushdown** `GR(R)` **bits** 32:63, `GR(X)`
3	`PUSHDNL` **pushdown** `GR(R),GR(X)`
5	`PUSHDNS` **pushdown** `SR(R),GR(X)`
6	`PUSHDNK` **pushdown** `KR(R),GR(X)`
7	`PUSHDND` **pushdown** `DR(R),GR(X)`
8	`POPUPB` **popup** `GR(R)` **bits** 56:63, `GR(X)`
9	`POPUPH` **popup** `GR(R)` **bits** 48:63, `GR(X)`
A	`POPUPW` **popup** `GR(R)` **bits** 32:63, `GR(X)`
B	`POPUPL` **popup** `GR(R),GR(X)`
D	`POPUPS` **popup** `SR(R),GR(X)`
E	`POPUPK` **popup** `KR(R),GR(X)`
F	`POPUPD` **popup** `DR(R),GR(X)`

TABLE 7.4A: THE NATURAL STACK/REGISTER COPYING INSTRUCTIONS

The actions '**pushdown** *r*, *x*' and '**popup** *r*, *x*' are defined as follows, where sizeof(*r*) denotes the size of the actual operand: 1, 2, 4 or 8 bytes:

pushdown *r*, *x*	*x* := *x* – sizeof(*r*); MAR := *x*; @MAR := *r*;

popup *r*, *x*	MAR := *x;*
	x := *x* + sizeof(*r*);
	r := @MAR;

The tertiary instructions G=C; T=B; provide stack push and pop operations for all register sets and stacks oriented in the 'inverse' direction, i.e. upward. They are as shown in Table 7.4B.

E	Operation
0	PUSHUPB **pushup** GR(R) **bits** 56:63, GR(X)
1	PUSHUPH **pushup** GR(R) **bits** 48:63, GR(X)
2	PUSHUPW **pushup** GR(R) **bits** 32:63, GR(X)
3	PUSHUPL **pushup** GR(R), GR(X)
5	PUSHUPS **pushup** SR(R), GR(X)
6	PUSHUPK **pushup** KR(R), GR(X)
7	PUSHUPD **pushup** DR(R), GR(X)
8	POPDNB **popdown** GR(R) **bits** 56:63, GR(X)
9	POPDNH **popdown** GR(R) **bits** 48:63, GR(X)
A	POPDNW **popdown** GR(R) **bits** 32:63, GR(X)
B	POPDNL **popdown** GR(R), GR(X)
D	POPDNS **popdown** SR(R), GR(X)
E	POPDNK **popdown** KR(R), GR(X)
F	POPDND **popdown** DR(R), GR(X)

TABLE 7.4A: THE INVERSE STACK/REGISTER COPYING INSTRUCTIONS

The actions '**pushup** *r*, *x*' and '**popdown** *r*, *x*' are defined as follows:

pushup *r*, *x*	MAR := *x*; *x* := *x* + sizeof(*r*); @MAR := *r*;

popdown *r*, *x*	*x* := *x* - sizeof(*r*);
	MAR := *x*;
	r := @MAR;

M7.5 Conditional copy (select) instructions

C	0	SELSBR	GR(R) := T(**if** BTR **then** (*n*+*x*) **else** GR(Y)), i.e. source **bits** 0..7, zero filled
C	1	SELSHR	GR(R) := T(**if** BTR **then** (*n*+*x*) **else** GR(Y)), i.e. source **bits** 0..15, zero filled
C	2	SELSWR	GR(R) := T(**if** BTR **then** (*n*+*x*) **else** GR(Y)), i.e. source **bits** 0..31, zero filled
C	3	SELSLR	GR(R) := **if** BTR **then** (*n*+*x*) **else** GR(Y)
C	4	SELUBR	GR(R) := T(**if** BTR **then** (*n*+*x*) **else** GR(Y)), i.e. source **bits** 0..7, sign extended
C	5	SELUHR	GR(R) := T(**if** BTR **then** (*n*+*x*) **else** GR(Y)), i.e. source **bits** 0..15, sign extended
C	6	SELUWR	GR(R) := T(**if** BTR **then** (*n*+*x*) **else** GR(Y)), i.e. source **bits** 0..31, sign extended
C	7	SELULR	GR(R) := **if** BTR **then** (*n*+*x*) **else** GR(Y)
C	D	SELDR	DR(R) := **if** BTR **then** DR(X) **else** DR(Y)
T	E	mnemonic	effect

TABLE 7.5A: THE SELECT INSTRUCTIONS

M8 THE GROUP D DIGIT SHIFT (*etc*) AND DEBUGGING INSTRUCTIONS

M8.1 Digit shift instructions

Instructions of the shift category are in the tertiary format. They operate on the general registers. General register Y is the source, containing the operand to be shifted. General register R is the destination for the result. T specifies the operand size and E encodes the operation. ($n + x$) determines the size of the shift.

There are three kinds of shift instruction.

- ARITHMETIC shifts, `S{LR}A{BHWL}`, treat the source operand as a signed integer, so that left shifts may overflow, and right shifts propagate the sign bit into the vacated places. Overflow when shifting left causes the Operand/Result Out Of Range trap.
- ROUNDED ARITHMETIC shifts, `SRR{BHWL}`, treat the source operand as a signed number, propagating the sign bit into the vacated places. The value this yields is then rounded by adding 1 if its last shifted-out bit was a 1. (These instructions are useful with fixed-point values that have a fractional part.)
- LOGICAL shifts, `S{LR}L{BHWL}`, move all the source bits the given number of places left or right, the vacated places being filled by zero-bits and the bits moved out of the other end being lost.
- CIRCULAR 'rotate' shifts, `S{LR}C{BHWL}`, move all the source bits the given number of places left or right, the vacated places being filled by the bits moved out of the other end.

Each of these is provided in byte, halfword, word and longword versions. The byte, halfword and word versions take as their source the least significant 8/16/32 bits of source register Y, shifting them within a field of 8/16/32 bits, before extending the result appropriately back to 64 bits.

0	`SLAB` shift byte left arithmetic	`SLAH` shift halfword left arithmetic	`SLAW` shift word left arithmetic	`SLAL` shift longword left arithmetic
1	`SRAB` shift byte right arithmetic	`SRAH` shift halfword right arithmetic	`SRAW` shift word right arithmetic	`SRAL` shift longword right arithmetic
2	`SLLB` shift byte left logical	`SLLH` shift halfword left logical	`SLLW` shift word left logical	`SLLL` shift longword left logical
3	`SRLB` shift byte right logical	`SRLH` shift halfword right logical	`SRLW` shift word right logical	`SRLL` shift longword right logical
4	`SLCB` shift byte left circular	`SLCH` shift halfword left circular	`SLCW` shift word left circular	`SLCL` shift longword left circular
5	`SRCB` shift byte right circular	`SRCH` shift halfword right circular	`SRCW` shift word right circular	`SRCL` shift longword right circular
6	`SRRB` shift byte right rounded	`SRRH` shift halfword right rounded	`SRRW` shift word right rounded	`SRRL` shift longword right rounded
E	T= 0, byte	T = 1, halfword	T = 2, word	T = 3, longword

TABLE 8.1A: THE SHIFT INSTRUCTIONS

Special considerations apply when ($n + x$) = *the operand size* (i.e. 8, 16, 32 or 64 bits):

- An arithmetic shift to the left causes the Operand/Result Out Of Range trap if the operand is not zero.
- An arithmetic shift to the right yields a result of -1 when the operand is negative, otherwise 0.
- A rounded arithmetic shift to the right always yields a result of 0.
- A logical shift always yields a result of 0.
- A circular shift yields the operand unchanged.

If ($n + x$) < 0 or ($n + x$) > *the operand size* (i.e. 8, 16, 32 or 64, respectively) then the ORR trap is caused.

M8.2 Single-digit setting and testing instructions

Instructions of this category are in the tertiary format. They operate on registers from any of the general, special or kernel banks. $(n + x)$ determines the bit number to be inspected and/or changed.

0	CLRBTG	CLRBTS	CLRBTK	Copy register Y of bank T−4 to register R of bank T−4, clearing bit $(n + x)$ of the result.
1	SETBTG	SETBTS	SETBTG	Copy register Y of bank T−4 to register R of bank T−4, setting bit $(n + x)$ of the result.
2	INVBTG	INVBTS	INVBTK	Copy register Y of bank T−4 to register R of bank T−4, inverting bit $(n + x)$ of the result.
3	TSTBTG	TSTBTS	TSTBTK	BTR, GR(R) := (register Y of bank T−4 **and** $2^{(n + x)}$).
4	TSTCLG†	TSTCLS	TSTCLK‡	BTR, GR(R) := (register Y of bank T−4 **and** $2^{(n + x)}$), and **c**lear bit $(n + x)$ of register Y.
5	TSTSTG†	TSTSTS	TSTSTK‡	BTR, GR(R) := (register Y of bank T−4 **and** $2^{(n + x)}$), and **s**et bit $(n + x)$ of register Y.
6	TSTNVG†	TSTNVS	TSTNVK‡	BTR, GR(R) := (register Y of bank T−4 **and** $2^{(n + x)}$), and i**nv**ert bit $(n + x)$ of register Y.
E	T= 4	T = 5	T = 6	Effect

TABLE 8.2A: THE SINGLE-BIT INSTRUCTIONS

If $(n + x) < 0$ or $(n + x) > 63$, then the ORR trap is caused.

† A TSTCLG, TSTSTG, or TSTNVG instruction with R = Y overwrites GR(R) with the updated GR(Y) value, not with a copy of the value placed in BTR.

‡ The TSTCLK, TSTSTK, and TSTNVK instructions test and update KR(Y) **atomically**. That is to say, no other update of that register is possible between reading its value and writing back its new value. This is important, for example, in enabling us to clear an interrupt request flag in the PDI register without risking a race condition on its other bits being set by new interrupt requests.

M8.3 Debugging (trap) instructions

Instructions of the trap category have T∈ [A, F], and effect a subroutine call to a fixed location if and only if certain conditions obtain. A trap order does not change the flow of control unless the corresponding trap-enable bit is set in the Trap Enable Register (TER). The address of the instruction following a taken trap is saved in the TRA register (SR8) and the trap number is saved in the ENR register (SR6). A taken trap jumps to the address given by TVR+8*i, where i is the trap number, and TVR is the Trap Vector Register (see §M1.2.1).

T	mnemonic	usage	effect
A	TRAPA	TRAP ON ADDRESS Trapped addresses designate locations that are possibly non-existent, being between the top of the heap and the top of the stack. The X and N fields are unused and RFE.	**if** TER(*ARE*) **and then** GR(R) > HLT **and then** GR(R) < SLT **then** ENR := *ARE*; TRA := IAR; IAR := TAR + 8**ARE*; **end if;**
B	TRAPB	TRAP ON SIGNED BOUNDS Values, intended for example as array or case indexes, can be checked before a potentially catastrophic use.	**if** TER(*SRE*) **and then** GR(R) **not in signed** $(n + x)$..GR(Y) **then** ENR := *SRE*; TRA := IAR; IAR := TAR + 8**SRE*; **end if;**
C	TRAPP	TRAP ON POINTER Addresses can be verified as pointing into a range of locations, allowing more fine grained checking than is provided by TRAPA.	**if** TER(*URE*) **and then** GR(R) **not in unsigned** $(n + x)$..GR(Y) **then** ENR := *URE*; TRA := IAR; IAR := TAR + 8**URE*; **end if;**
D	TRAPD	TRAP UNLESS DISABLED Can be used to flag up entry to stub code, for example, or to establish a breakpoint.	**if** TER(*UST*) **then** ENR := $(n + x)$; TRA := IAR; IAR := TAR + 8**UST*; **end if;**
E	TRAPT	TRAP ON TRUE See below.	**if** TER(*TVT*) **and then** GR(R) ≠ 0 **then** ENR := $(n + x)$; TRA := IAR; IAR := TAR + 8**UST*; **end if;**
F	TRAPF	TRAP ON FALSE See below.	**if** TER(*FVT*) **and then** GR(R) = 0 **then** ENR := $(n + x)$; TRA := IAR; IAR := TAR + 8**UST*; **end if;**

TABLE 8.3: THE TRAP INSTRUCTIONS

The TRAPT and TRAPF instructions are useful in debugging and in diagnostic code, ensuring that the logical precondition for some action is met. The values 1 and 0 are the respective representations of True and False as generated by the 'test' / 'compare' instructions, and can therefore be tested by the TRAPT and TRAPF orders; but note that TRAPT considers **any** non-zero value as True. See Sections M7.1, M7.2 and M7.5.

M8.4 Debugging (OBEY) instruction

The OBEY instruction, T=8, is in the primary format. The R, X and N fields are unused and RFE.

Special register OIS (SR7) determines the address of the instruction to be obeyed. When the instruction execution cycle for the addressed instruction is complete, OIS is set to the address that would have been left in IAR had the OBEY-ed instruction been executed in the course of normal control flow. Control returns to the instruction following the OBEY instruction, unless an unhandled trap is caused by the obeyed instruction. If it causes a trap that is handed-off, the value left in OIS is the address of the vectored trap routine.

It is illegal, and causes a ROA (Recursive OBEY Attempt) trap, to execute an OBEY order via an OBEY order.

The OBEY order is useful in implementing a tracing facility in a debugger, since it permits the latter to single-shot the code being debugged, with no danger of it running out of control.

M9 GROUP E IS ENTIRELY RESERVED FOR EXTENSIONS OF THE ARCHITECTURE

Here is an example of the kind of thing that might be done in group E.

The supplementary modifier orders, SMOAW Re Ne[Xe], and SMOAL Re Ne[Xe]:

N_e	X_e	R_e	G = E	T = E or T = F

augment the immediate operand or stored-operand address of the following order, its **extended order**. It is invalid and causes the Reserved Instruction Format trap if the extended order is not an appropriate primary-format or secondary-format order with an immediate or stored operand.

The effect of SMOAW (T = E) is:

a. for a primary-format extended order: prepend the N_e field to its N_p-part, yielding a 32 bit value, μ

b. for a secondary-format extended order: prepend both the N_e field and the X_e field to its N_s-part, yielding a 32 bit value, μ

c. sign-extend μ to 64 bits, a

d. if $R_e \neq 0$: specify GR(R_e) as a base address register with contents of value b

e. if $R_e = 0$: specify that $b = 0$

f. compute an immediate operand as: $b + a + x$, where x denotes the index value specified in the extended instruction

g. compute a stored-operand address as: $b + a + x^*$, where x^* denotes the (possibly scaled, see §M2.2) index value specified in the extended instruction

The effect of SMOAL (T = F) is:

a. for a primary-format extended order: prepend the N_e field to its N_p-part, yielding a 32 bit value, μ

b. for a secondary-format extended order: prepend both the N_e field and the X_e field to its N_s-part, yielding a 32 bit value, μ

c. prepend μ to the 32-bit word immediately following the extended order, thus lengthening its value to 64 bits, a

d. if $R_e \neq 0$: specify GR(R_e) as a base address register with contents of value b

e. if $R_e = 0$: specify that $b = 0$

f. compute an immediate operand as: $b + a + x$, where x denotes the index value specified in the extended instruction

g. compute a stored-operand address as: $b + a + x^*$, where x^* denotes the (possibly scaled, see §M2.2) index value specified in the extended instruction

M10 THE GROUP F (FLOATING POINT, FRAME-HANDLING, *etc*) INSTRUCTIONS

Group F instructions are first determined by T, as shown in Table 10.0A.

T	Format	Operation
0	primary	STSKC
1	primary	LDLNK
2	primary	LEAVE
3	primary	RETS
4	primary	RETL
5	primary	TRIM
6	N/A	RFE
7	primary	RETT
8-B	secondary	floating point arithmetic operations
C	primary	SYSC
D	primary	DOIO
E	primary	ENTER
F	primary	FRAME

TABLE 10.0A: GROUP F INSTRUCTION FORMATS

M10.1 Interlocking store and load instructions for inter-process synchronization

These instructions are used to implement exclusive access to shared longword variables in critical sections of concurrent programs (such as the kernel). They are executed **atomically**: that is, they either complete, or they have no effect (if interrupted in any manner).

T	Mnemonic	Operation	Effect

0	STSKC	**st**ore and **sk**ip **c**onditional	**let** EA = (*n* + *x*);
			if EA **is reserved then**
			  @EA := GR(R);
			  IAR := IAR + 4;
			end if;

1	LDLNK	**l**oa**d** **link**ed	**let** EA = (*n* + *x*); **reserve** EA; GR(R) := @EA;

TABLE 10.1A: THE INTERLOCKING INSTRUCTIONS

The action **reserve** EA is implemented by copying the value of EA into the Linked Address Register (LAR) and setting a flag to indicate that LAR *is significant*. The condition (EA **is reserved**) is implemented by evaluating the expression (LAR *is significant* **and** LAR=EA). The linkage is broken—by unsetting the flag, or by changing the value held in LAR—as a result of any event that might change the value held in the location @LAR, or whose effect on the value held in the location @LAR cannot be predicted by the processor. These events include STSKC instructions, LDLNK instructions, store instructions, interrupts and context switches; and might vary between different realizations of the M architecture.

M10.2 The (stack) frame-handling instructions

These instructions are used to administer the STACK FRAME (ACTIVATION RECORD) for procedures and functions (or interrupts). They enable a variety of calling protocols that may be tailored, for minimum overhead, to the properties of both the calling and the called routines. The specimen code sequences given here illustrate the power, flexibility and efficiency of these mechanisms in the M architecture.

Figure 10.2A is a picture of a general stack frame, immediately after completing entry to a subroutine.

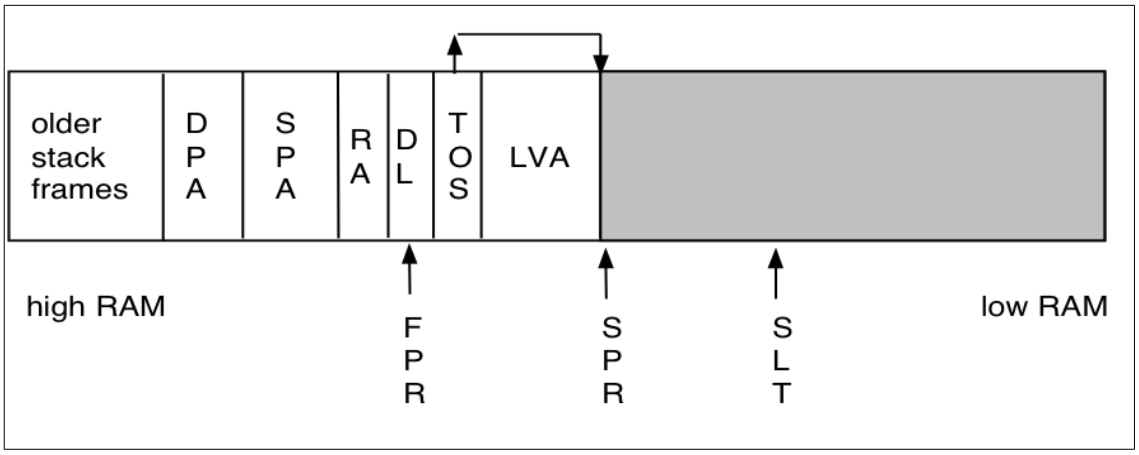

FIGURE 10.2A: STANDARD STACK-FRAME LAYOUT

The stack grows from high addresses to lower addresses. A stack frame may contain any of the following components, each of which may be omitted from the frame if not needed:

- DPA—the DYNAMIC PARAMETER AREA, for actual parameters (such as dynamic arrays) whose size is not known at compile time.
- SPA—STATIC PARAMETER AREA, for actual parameters whose size is known at compile time, (including pointers into the DPA to locate the start of each dynamic parameter).
- RA—RETURN ADDRESS, the value put in `IAC` by the `CALL` instruction and saved in the stack by the `ENTER` instruction.
- DL—DYNAMIC LINK, the value of `FPR` in the calling routine, as saved by the `ENTER` instruction. If a STATIC LINK is needed it may be passed in `EPR` and/or in the last word of the SPA.
- TOS—TOP OF STACK, a pointer to the first unallocated stack location, as saved by the `FRAME` instruction (used to reset `SPR` after nested calls).
- LVA—LOCAL VARIABLE AREA, static size locals first, followed by dynamically sized locals. If local allocators are evaluated during the execution of the subprogram, they take space from the top of the stack, abutting the LVA, and adjusting `SPR` and TOS accordingly.

The subprogram calling mechanisms use four of the special registers:

C	`SLT`	Stack LimiT register
D	`SPC`	Stack Pointer Copy register
E	`FPC`	Frame Pointer Copy register
F	`IAC`	Instruction Address Copy register
#	Mnemonic	Special functionality

TABLE 10.2A: SPECIAL REGISTERS USED BY CALLING MECHANISM

and the instructions described in Table 10.2B; in addition to the `BASRA` and `CALL` instructions described in §6.

2	LEAVE	**ret**urn from **r**ecursively-general subroutine, restoring IAR and FPR, and making the **r**esidual frame addressable via EPR	EPR := FPR; IAR := @(FPR+8); FPR := @FPR;
3	RETS	**ret**urn from **s**imple subroutine with **s**tack **s**pace, restoring SPR and IAR	SPR := SPC; IAR := IAC;
4	RETL	**ret**urn from **l**eaf subroutine, restoring IAR	IAR := IAC;
5	TRIM	trim the stack to the given end-point	**let** *TOS* = @(*n* + *x*); **trap** SLV **unless** *TOS* ≥ SPR **and** *TOS* ≥ SLT **and** *TOS* < FPR; GR(R) := *TOS*;
7	RETT	**ret**urn from **t**rap, restoring interrupted context	ENR := 0; ESF := ESF - {*trap_state*}; IAR := TRA;
E	ENTER	save linkage to calling context's code and stack frame	**align** SPR **to longword**; @SPR := IAC; SPR := SPR – 8; FPR := SPR; @SPR := FPC; SPR := SPR – 8;
F	FRAME	establish called frame enclosing the last local variable	**let** *TOS* = (*n* + *x*); **align** *TOS* **to longword**; @SPR := *TOS*; SPR := SPR – 8; SPR := *TOS*;
T	Mnemonic	Operation	Effect

TABLE 10.2B: THE LEAVE, RETS, RETL, TRIM, RETT, ENTER AND FRAME INSTRUCTIONS

Calling, entry and exit sequences

The following instruction sequences have been devised with the intention of fully supporting the semantics of procedures and functions in languages as powerful as Ada, while allowing simple cases to be given simple treatment. Throughout, instructions or instruction sequences may be omitted if some part of the general stack frame is not needed for a particular routine.

Parameters may be passed in registers, or in the stack; or both. By convention, a function result is returned in register 1 of the appropriate (general or floating point) set. If it is too large for a register, or if it has dynamic size, its size is left in GR0 and its address is left in GR1.

A subprogram is invoked by a **calling sequence**, that passes parameters, jumps to the subprogram (see the BASRA and CALL instructions, §6), deals with output parameters and function results, and finally restores the stack frame of the calling routine.

Many subprograms are simple and their calling sequences may be tailored to their lesser requirements, always conforming to (some subset of) the standard stack frame layout. In the simplest case no stack frame need be allocated at all.

Here is the general case:

1. Push any in-stack parameters of dynamic size (if any).
2. Push any in-stack parameters of static size; including the static link, as needed.
3. Load any in-register parameters; including the static link to EPR, as needed.
4. CALL FPR *subr*
5. Save any result parameters, using EPR to address the residual stack frame, as needed.
6. Restore SPR, to pop the residual stack frame, as needed.

A convenient and safe way to pop the residual stack frame is provided by the TRIM instruction. It checks its operand for validity. If it is valid it is saved in GR(R), which will normally be the SPR register, GRD. Used with -8[FPR] (the top-of-stack pointer stored in the stack frame) it combines a trimming action with some validation of TOS, which might have been corrupted by errant code in a nested subprogram call.

Each routine begins with an **entry sequence**, to establish its execution context; and ends with an **exit sequence**, to restore the execution context of the calling routine. In the general case, the full stack frame layout shown in Figure 10.1 needs to be set up on entry and torn down on exit. ENTER works with FRAME to set up stack frames. LEAVE, RETS, and RETL all exit to the calling context; LEAVE also restores its stack frame.

The general-case (fully recursive) entry sequence with *S* bytes of static local data (including the TOS value) and *D* bytes of dynamic local data is as follows:

```
ENTER                          | push saved stack pointers and set new frame pointer
calculate SPR-D in GR(R)       | reserve D bytes for dynamic locals
FRAME   S[GR(R)]
```

The fully recursive entry sequence with *S* bytes of static local data only is:

```
ENTER                          | push saved stack pointers and set new frame pointer
FRAME   S[SPR]                 | omit entirely if both S and D are 0
```

The general-case exit sequence is simply:

```
LEAVE                          | restore IAR and FPR from stacked link
```

The calling, entry and exit sequences can be considerably less elaborate, depending on the properties of the calling and (more strongly) the called routine. If the called routine is a LEAF (one that itself calls no further subroutines) and/or if it does not allocate stack space for local in-stack data, major simplifications can be made. Using CALL SPR *subr* in the calling sequence ensures that IAC and SPC contain information adequate to restore the calling context, so that ENTER and FRAME operations are unnecessary, and the RETS (or RETL) operation suffices to return.

The leaf entry sequence is as follows:

```
calculate D in GR(R)           | reserve D bytes for dynamic locals, if needed
SUBUWI SPR S[GR(R)]            | or S[GR0], if D is 0; omit entirely if both S and D are 0
```

The leaf exit sequence is:

```
RETS                           | restore IAR and SPR from IAC and SPC, as needed
```

or even (if there is no local data):

```
RETL                           | restore IAR from IAC
```

Note that a leaf routine with no in-stack local data incurs the smallest possible overhead of 2 instructions in total for the calling, entry and exit sequences.

An alternative protocol for the latter case is to use BASRA GR(R) *subroutine* (= GOSUB *subroutine*) to invoke the routine and BASRA IAR 0[GR(R)] (= GOBACK 0[GR(R)]) to return. This avoids over-writing the contents of IAC and SPC, enabling a further optimization. Namely, that a non-leaf routine calling only leaf routines using the GOSUB protocol need not save IAC and SPC, allowing it to use the leaf entry and exit sequences itself.

Stack limit checking

If the completion of an ENTER, FRAME, LEAVE, or RETS instruction would set SPR or FPR to a value less than SLT, the SLV trap is caused, just as it is by the TRIM instruction.

M10.3 Floating point memory reference instructions

These are secondary-format instructions involving an expression of the form `DR(R)` *op* T(V), with the operation determined by the E field of the instruction. Non-register operands are 4 bytes in length (E format, single-precision), or 8 bytes in length (D format, double-precision), as determined by T. A floating point stored operand has an address that is a multiple of its length.

0	`STREI`	`STRE`	`STRDI`	`STRD`
1	`LODEI`	`LODE`	`LODDI`	`LODD`
2	`ADDEI`	`ADDE`	`ADDDI`	`ADDD`
3	`SUBEI`	`SUBE`	`SUBDI`	`SUBD`
4	`MULEI`	`MULE`	`MULDI`	`MULD`
5	`DIVEI`	`DIVE`	`DIVDI`	`DIVD`
6	`FLREI`	`FLRE`	`FLRDI`	`FLRD`
7	`RNDEI`	`RNDE`	`RNDDI`	`RNDD`
8	`NEGEI`	`NEGE`	`NEGDI`	`NEGD`
9	`CLGEI`	`CLGE`	`CLGDI`	`CLGD`
E	T=8	T=9	T=A	T=B

TABLE 10.3A: THE FLOATING POINT INSTRUCTIONS

The OPERAND VALUE T(V) is determined by the operand size, as shown in Table 10.3B.

float $(n + x)$	`@(EA..EA+3)`	**double** $(n + x)$	`@(EA..EA+7)`
T=8	T=9	T=A	T=B

TABLE 10.3B: T(V) **FOR THE FPU OPERATIONS OF TABLE 10.3A**

A floating point store immediate instruction uses the R field to select one of a table of constants held within the CPU, and stores that in the operand addressed by $(n+x)$. These constants are:

0	10.0	
1	1.0	
2	2.0	
3	3.0	
4	4.0	
5	5.0	
6	*e*	
7	π	
8	+Inf	`7FF0_0000_0000_0000`
9	positive signalling NaN	`7FF0_0000_0000_0001`
10	positive quiet NaN	`7FF8_0000_0000_0000`
11	-0.0	`8000_0000_0000_0000`
12	-Inf	`FFF0_0000_0000_0000`
13	negative signalling NaN	`FFF0_0000_0000_0001`
14	Indeterminate	`FFF8_0000_0000_0000`
15	negative quiet NaN	`FFF8_0000_0000_0001`
R	Value	Bit pattern

TABLE 10.3C: THE STORABLE IMMEDIATE CONSTANTS

Zero can be stored with the `STR{SU}WI` and `STR{SU}LI` instructions.

M10.4 The SYSC (system call) instruction

The Operating System is invoked by a process from user state with a `SYSC` instruction, which causes a trap to the kernel. The operand is given by the value $(n + x)$, the SYSCALL NUMBER, which identifies the operation to be performed for the process by the kernel.

For further implementation on the system calls presently supported by **mekhos**, see §K.

M10.4 The DOIO instruction

The instruction `DOIO` (**do i**nput/**o**utput) is PRIVILEGED—that is, it can be executed to completion only in real kernel state. An attempt to execute `DOIO` in a user state causes the User-State Privilege Violation `(UPV)` trap. An attempt to execute `DOIO` in virtual kernel state causes the Virtual Machine Monitor `(VMM)` trap.

The operand of `DOIO`, given by the value $(n + x)$, is the effective address of a group of 64 bytes in RAM, which contain a CONTROLLER CONTROL BLOCK (CCB) defining a command to the I/O subsystem. `GR(R)` is set to the I/O device's response to that command: 0 if the command was accepted; and a non-zero code for a rejected command. Response codes may be device-specific, and are implementation defined; however a code of 0 always indicates a successfully initiated operation.

The order in which the parameters are checked is implementation defined. If an error is detected in any parameter, any other parameters need not be checked and may contain undetected errors.

The following layout for the Controller Control Block is provisional:

- longword 0: **bits** 0..15: command; **bits** 16:31: option flags; **bits** 32:63: device number
- longword 1: status at end of transfer
- longword 2: transfer size
- longword 3: RAM address of buffer
- longword 4: device parameter 0 (e.g. disk drive number)
- longword 5: device parameter 1 (e.g. disk cylinder number)
- longword 6: device parameter 2 (e.g. disk track number)
- longword 7: device parameter 3 (e.g. disk block number)

End-of-transfer status codes may be device-specific, and are implementation defined; however a code of 0 always indicates a successfully completed operation.

M11 IMPLEMENTATION DEFINED CHARACTERISTICS

The architectural specification above does not fix the means by which M programs are prepared, loaded and run. The following sections specify A, the assembly-language translator; E, the M emulator; L, the loadable program format; and K, the emulated Operating System kernel.

These software components constitute the **mekhos** implementation of the M architecture. Its implementation defined and implementation dependent characteristics are as follows.

M11.1 macOS implementation of mekhos

- Invalid floating point operations generate IEEE NaN or Inf values, and do not cause an ORR trap.
- Fetching the Calendar-Clock Register or the Nanoseconds Register makes the emulator obtain the current date and/or time from the host operating system, and so is much slower than other M instructions. For this reason, on termination of a run after only a small number of instructions have been obeyed, the value in the Nanoseconds Register is not likely to be useful.

A

THE ASSEMBLY LANGUAGE

A1 INTRODUCTION AND SYNTAX

The A language has three distinct kinds of element: *data declarations*, *machine instructions*, and *directives*. The latter are commands to the assembler, telling it how to handle the other elements.

An A *program* is a sequence of *routines*; each routine is headed by a directive and contains a mix of machine instructions and data declarations. The program starts executing at the first instruction of the routine introduced by the ***PROGRAM** directive.

Data declarations are of five kinds: *argument* and *local* declarations, which associate symbolic names with offsets in a routine's stack frame; *global* declarations, which name, reserve and optionally initialize variable data of global scope and global lifetime; *pool* declarations, which name, reserve and initialize constant data of global lifetime; *static* declarations, which name, reserve and optionally initialize variable data of local scope and global lifetime; and *external* declarations, which reference global data declared elsewhere.

End-of-line comments may appear, introduced by a vertical bar. A comment introduced by a pair of vertical bars is transcribed into the L object program. Blanks and tabs may be used at will between code elements, and empty lines may be used at will to format the code for readability.

Identifiers, numbers and *strings* have the same syntax as in L. The letter case of directives, types, register names, and instruction mnemonics is not significant; but for good style they should be spelled consistently, either in all-upper case or in all-lower case. The style preferred here is to write mnemonics in lower case and other reserved words in upper case. Letter case **is** significant and preserved in declared identifiers.

DIRECTIVES

***ROUTINE** *identifier*

This introduces a name for, and delimits the start of, a new scope.

***PROGRAM** *identifier*

This has the same effect as the ***ROUTINE** *identifier* declaration, but also nominates this routine as the program entry point; see § L6.

***END** [*identifier*]

This delimits the end of the current scope. If the optional *identifier* is given, it must match that scope's name.

DECLARATIONS

***POOL** *declarator* **=** *initializer*

This introduces a name for, and reserves, an area in the pool segment; see §L2. The name has local scope if the directive occurs within a routine, and global scope otherwise. The initializer specifies constant value(s) to be placed in the area when the program is loaded. They should not be updated by the running program and, in an implementation of M that includes memory protection, should have read-only access status.

***GLOBAL** *declarator* [**=** *initializer*]

This introduces a name for, and reserves, an area in the data segment; see §L2. The name has global scope regardless of the position of the directive. The optional initializer specifies any constant value(s) to be placed in the area when the program is loaded. They may be updated by the running program and, in an implementation of M that includes memory protection, have read-write access status.

***STATIC** *declarator* [**=** *initializer*]

This introduces a name for, and reserves, an area in the data segment; see §L2. The name is local to the current scope. The optional initializer specifies any constant value(s) to be placed in the area when the program is loaded. They may be updated by the running program and, in an implementation of M that includes memory protection, have read-write access status.

***EXTERNAL** *declarator*

This allows (forward) reference to a name that is declared, as a global or a routine, in another module.

***ARGUMENT** *declarator*

This introduces a name for, and reserves a location within, the scope's parameter area; see Figure 10.2A.

***LOCAL** *declarator*
This introduces a name for, and reserves a location in, the scope's local variable area; see Figure 10.2A.

***NUMBER** [**BYTE** | **HALF** | **WORD** | **LONG** | **FLOAT** | **DOUBLE**] *identifier* **=** *number*
This solely gives a name to a constant number; it reserves no storage. The name may be used instead of the number in any place that a number is required (e.g. in an *initializer*). The name has local scope if the directive occurs within a routine, and has global scope otherwise.

Label '**:**'
This introduces an identifier for a location in the code segment; its value is the address of the following *machine_instruction*. A label declared in this way is local to, and unique within, the current scope.

**decimal_digit*
This introduces a Knuth-style temporary label; its value is the address of the following *machine_instruction*. A usage preceding this directive refers to the label by the name *decimal_digit*`f`, e.g. `9f`; a usage following this directive refers to the label by the name *decimal_digit*`f`, e.g. `9b`. Further occurrences of the directive declare new labels, reusing that name for a new value.

DECLARATORS
In the following *declarators*, the optional *length* specifies the size of the area being declared, as a multiple of the specified operand type; in its absence, one location of the type is allocated.

BYTE *identifier* [*length*]

HALF *identifier* [*length*]

WORD *identifier* [*length*]

LONG *identifier* [*length*]

FLOAT *identifier* [*length*]

DOUBLE *identifier* [*length*]

For the `STRING` *declarator*, the size of the area to be allocated is determined by the specified initial value.

STRING *identifier*

LENGTHS
'`[`' *number* '`]`'

INITIALIZERS
for **STRING**: *string*
for **BYTE**: $b_0 \ldots b_n$
for **HALF**: $h_0 \ldots h_n$
for **WORD**: $w_0 \ldots w_n$
for **LONG**: $l_0 \ldots l_n$
for **FLOAT**: $f_0 \ldots f_n$
for **DOUBLE**: $d_0 \ldots d_n$
where $b_0 \ldots d_n$ are *operand*s of appropriate type and magnitude.

Orders

Label ‘**:**’ *machine_instruction*

This inserts an executable instruction into the code segment and introduces a label for its location. A label declared in this way is local to, and unique within, the current scope.

machine_instruction

This inserts an unlabelled executable instruction into the code segment.

Labels

identifier

Machine instructions

mnemonic register operand [*indexer*]	immediate/stored operand formats
mnemonic	implicit operand format
mnemonic register	1 register format
mnemonic register register	2 register format
mnemonic register register ‘**,**’ *operand* [*indexer*]	3 register format

Registers

kr*hex_digit* |
ESF | **AVR** | **IDN** | **PDI** |
LAR | **IRA** |
CCR | **NSR** | **ICR** | **TRA** |

sr*hex_digit* |
QMD | **BTR** | **ENR** | **TER** |
OIS | **TVR** | **RWF** | **HLT** |
SLT | **SPC** | **FPC** | **IAC** |

gr*hex_digit* |
EPR | **SPR** | **FPR** | **IAR** |

dr*hex_digit*

Indexers

‘**[**’ **gr***hex_digit* [‘**<<**’ *decimal_digit*] ‘**]**’

The *decimal_digit* indicates scaling of the value in the index register; it must lie in the range 0..1 for a halfword operand, in 0..3 for a word operand, and in 0..7 for a doubleword operand.

Operands

[‘**+**’ | ‘**-**’] *number*

The operand is the value of the appropriately signed *number*.

[*relativiser*] *identifier*

The operand is the address of the *identifier*.

[*relativiser*] *identifier* [‘**+**’ | ‘**-**’] *number*

The operand is the address of the *identifier*, offset by the specified number.

[*relativiser*] *digit_label*

The operand is the current definition of the *digit_label*.

‘**(**’ *operand* ‘**)**’

The operand is the value of the enclosed operand, in one of the above forms.

= *literal*

The operand is a location in the constant pool that has been automatically set to the specified *literal*.

Relativisers

‘**$**’

A *relativiser* specifies program-relative addressing. No indexer should be given, as `IAR` is used automatically, but an explicit use of `IAR` is tolerated. The address put into the instruction is that of the operand, reduced by (4 + *the address of the current instruction*), which is the value in `IAR` when the instruction is executed.

LITERALS

number | *string*

DIGIT LABELS

decimal_digit`f` |
decimal_digit`b` |
decimal_digit`F` |
decimal_digit`B`

STRINGS

`'`*character_image*$^{+}$`'` or `"`*character_image*$^{+}$`"`

A string is a sequence of one or more character values, each represented by a *character_image*. Both `'` and `"` may be used as string quotes, so long as the opening and closing quotes are the same, but the chosen delimiter can not be given directly within the string. Examples: `'This is a null-terminated string\0'`, `"This is a line of output\n\0"`.

CHARACTER IMAGES

A *character_image* is either a graphic character of the Latin-1 set, or one of the following, representing an 8-bit value (the images `\'` and `\"` may be used to specify an apostrophe and a quote, respectively, in a string bounded by that character):

Image	Represents	Value in hex
`\0`	NUL	00
`\'`	apostrophe	27
`\"`	quote	22
`\\`	backslash	5C
`|`	vertical bar	7C
`\a`	**A**udible **a**larm (BEL)	07
`\b`	**B**ackspace (BS)	08
`\d`	**D**elete (DEL)	FF
`\f`	**F**orm Feed (FF)	0C
`\h`	**H**orizontal Tab (HT)	09
`\n`	**N**ew Line (NL)	0A
`\r`	Carriage **R**eturn (CR)	0D
`\v`	**V**ertical Tab (VT)	0B
`\x`*hd*	character with hex code *hd*	*hd*

NUMBERS

[`#`][`{0 | F | f}:`] {*hex_digit*}$^{+}$

The value is that of a (64-bit) hexadecimal number. If the '`{0 | F | f} :`' prefix is present, the more significant digits of the number preceding the given sequence, {*hex_digit*}$^{+}$, are set to the digit (0 or F) specified in the prefix. This provides a convenient abbreviation of negative numbers and global addresses.

For example, #F:0 equals #FFFFFFFFFFFFFFF0 and #0:3210 equals #0000000000003210

{*decimal_digit*}$^{+}$

The value is that of the decimal number.

'*character_image*' or "*character_image*"

The value is the Latin-1 code of the character depicted. Both ' and " may be used as string quotes, so long as the opening and closing quotes are the same, but the chosen delimiter can not be given directly within the string. Examples: 'a', '"', "'", '\n', '\'' , '\x0D'.

'*character_image character_image*' or "*character_image character_image*"

The value is (256* *the code of the first character depicted + the code of the second character depicted*). Both ' and " may be used as string quotes, so long as the opening and closing quotes are the same, but the chosen delimiter can not be given directly within the string. Examples: 'ab', '"″', "'′", '\n\0', '\′\″', '\x0D\x0A'.

DECIMAL DIGITS

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

HEX DIGITS

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | a | b | c | d | e | f

A2 EXAMPLE OF AN A PROGRAM

```
*NUMBER LONG   gm = 3
*POOL   LONG   calls_for_n_equal_10 = 44780245
*POOL   LONG   gn = 10
*POOL   STRING message1 = "Ackermann(3, \0"
*POOL   STRING message2 = ") = \0"
*POOL   STRING message3 = ".\n\n\0"
*POOL   STRING message4 = " ns/call\n\n\0"

*PROGRAM main

*EXTERNAL ROUTINE Put_Long
*EXTERNAL ROUTINE Put_String
*EXTERNAL ROUTINE ack

*BEGIN
   loduli   fpr   TOS
   strul    fpr   DL[fpr]
   loduli   spr   -24
   strul    spr   TOS[fpr]
   lodul    gr2   = #FFFFFFFFFF00000
   cpygs    slt   gr2
   loduli   gr1   message1
   gosub    gra   Put_String
   lodul    gr1   gn
   gosub    gra   Put_Long
   loduli   gr1   message2
   gosub    gra   Put_String
   loduli   gr7   gm
   lodul    gr8   gn
   cpysg    gr6   nsr
   call     fpr   ack
   cpysg    gr5   nsr
   subsli   gr5   0[gr6]
   loddi    dr1   0[gr5]
   lodsl    gr2   calls_for_n_equal_10
   loddi    dr2   0[gr2]
   divdr    dr3   dr1, dr2
   cpygg    gr1   gr0
   gosub    gra   Put_Long
   loduli   gr1   message3
   gosub    gra   Put_String
   rnddr    dr3   dr3
   condr    gr1   dr3
   gosub    gra   Put_Long
   loduli   gr1   message4
   gosub    gra   Put_String
   sysc           exit
*END

*ROUTINE ack
```

```
|| the argument m is in gr7, n in gr8
*LOCAL LONG m | n need not be stacked - tail recursion on n
*BEGIN
   enter
   frame          m[spr]
   bgtz     gr7   $m_nonzero
   lodsli   gr0   1[gr8]         | return n+1 if m = 0
   leave                         | 5 inst to here
m_nonzero:
   bgtz     gr8   $n_nonzero
   subsli   gr7   1
   lodsli   gr8   1
   call     fpr   ack
   trim     spr   TOS[fpr]       | return Ack(m-1, 1), if n = 0
   leave
n_nonzero:
   strsl    gr7   m
   subsli   gr8   1
   call     fpr   ack            | compute Ack(m, n-1)
   trim     spr   TOS[fpr]
   lodsl    gr7   m
   subsli   gr7   1
   cpygg    gr8   gr0
   call     fpr   ack
   trim     spr   TOS[fpr]       | return Ack(m-1, Ack(m, n-1))
   leave                         | 14 inst to here, mean 9.5
*END ack
```

The subsidiary routines `Put_String` and `Put_Long` are held in separate files, `puts.a` and `putl.a` :

```
*ROUTINE Put_String     | in puts.a
*BEGIN
   cpygg   gr2    gr1
   lodsli  gr1    1
   sysc           put_nts
   goback         0[gra]
*END Put_String
```

and

```
*ROUTINE Put_Long    | in putl.a
*BEGIN
   lodsli   gr3   1
   cpygg    gr2   gr1
   lodsli   gr1   1
   sysc           put_sld
   goback         0[gra]
*END Put_Long
```

A3 Running the assembler from the command line

The assembler can be invoked from the command line, thus:

a [-b | -c | -d | -l*f* | *file_argument*] …

where each *file_argument* is the name of a program file.

Flags invoke options that apply to program files named to their right in the argument list.

Examples:

```
a -b
a subr1.a -le subr2.a -ln main.a
a -b program.a
```

Any program file parameter must be the name of a file with `.a` extension, containing code in A format.

As a concession to users of command-line completion it is permissible to omit the final '`.a`' from a program file name, or even just the final '`a`', so that if the parameter ends with '`.`' the assembler will complete the name automatically.

Each such file is assembled in turn. Global declarations (including routine declarations) made in one file are carried over to the rest of the list. If there is no program file parameter, `program.a` is used by default, so the first and third examples behave in exactly the same way.

A file with `.l` extension is written for each parameter of A. It includes the A source code in the form of commentary appended to each line of L-format object code. So, the second example creates object files named `subr1.l`, `subr2.l`, and `main.l`.

Logging messages that record the progress of the assembly, and details of any errors encountered, are written to the file `a_log.txt`. The level of detail may be controlled by the -l*f* parameter, as follows:

-ls: **s**uppresses all non-diagnostic logging output

-ln: provides **n**ormal logging output

-le: provides **e**xtensive output, including the contents of the local or global symbol tables, as appropriate, plus tables mainly useful in diagnosing faults in *a* itself; the latter may be suppressed unless *a* was compiled in a testing build

The flag -b engages **b**rief mode, in which source code is not included (as commentary) in the `.l` file.

The flag -c engages **c**hecking mode, in which object code is not generated and no output is made to the `.l` file.

The flag -d engages **d**iagnostic mode, in which optional internal consistency checks are made. This is intended primarily for diagnosing errors in *a* itself; output will be suppressed unless *a* was compiled in a testing build

Here is a run of the assembler, processing the example program in §A2:

```
/Users/wf/Mekhos/Testing/ack: a ackw puts putl

Assembling 'ackw.a' at: 2017-02-12 00:44:48.93 ...
   Start of routine 'main' ...
   ... end of routine 'main'
   Start of routine 'ack' ...
   ... end of routine 'ack'
... 'ackw.a' was successfully assembled, with 1 warning.


Assembling 'puts.a' at: 2017-02-12 00:44:48.93 ...
   Start of routine 'Put_String' ...
   ... end of routine 'Put_String'
... 'puts.a' was successfully assembled.


Assembling 'putl.a' at: 2017-02-12 00:44:48.93 ...
   Start of routine 'Put_Long' ...
   ... end of routine 'Put_Long'
... 'putl.a' was successfully assembled.

Assembly succeeded: no error was detected.
```

With extensive logging requested we get:

```
/Users/wf/Mekhos/Testing/ack: a -le ackw

Assembling 'ackw.a' at: 2017-10-27 00:56:10.12 ...
   Start of routine main ...
==========================
The GLOBAL symbol table has 56 item(s) in 119 buckets, of which 48 are occupied.
==========================
  12:901010103734B0B6 main______________________________0000000000000000 IS ROUTINE CODE     LOCATION  SET AT LINE 9
  14:33B737A62FBA3AA8 Put_Long__________________________0000000000000000 IS ROUTINE EXTERNAL LOCATION  REF AT LINE 11
  16:101010103230B2B9 read______________________________0000000000000004 IS LONG    NUMBER   NAMED     PRESET CONSTANT
    +55044604856446F6 DEBUG_CODE________________________0000000000000400 IS LONG    NUMBER   NAMED     PRESET CONSTANT
  19:10101010101033 18 0f_______________________________0000000000000000 IS LABEL   CODE     LOCATION
  20:10101010101033 19 2f_______________________________0000000000000000 IS LABEL   CODE     LOCATION
    +D535460485647 4D DEBUG_SRS________________________0000000000002000 IS LONG    NUMBER   NAMED     PRESET CONSTANT
  21:101010101010331A 4f________________________________0000000000000000 IS LABEL   CODE     LOCATION
  22:98B2B3B0B9B9B2B6 message1__________________________0000000000000010 IS STRING  POOL     LOCATION  SET AT LINE 4
    +1010101010103 31B 6f_______________________________0000000000000000 IS LABEL   CODE     LOCATION
  23:101010101010331C 8f________________________________0000000000000000 IS LABEL   CODE     LOCATION
  26:90101010101031 18 1b_______________________________0000000000000000 IS LABEL   CODE     LOCATION
  27:90101010101031 19 3b_______________________________0000000000000000 IS LABEL   CODE     LOCATION
  28:901010101010311A 5b________________________________0000000000000000 IS LABEL   CODE     LOCATION
  29:901010101010311B 7b________________________________0000000000000000 IS LABEL   CODE     LOCATION
  30:901010101010311C 9b________________________________0000000000000000 IS LABEL   CODE     LOCATION
  31:9932B3B0B9B9B2B6 message2__________________________0000000000000021 IS STRING  POOL     LOCATION  SET AT LINE 5
  33:90101010101036B3 gm________________________________0000000000000003 IS LONG    NUMBER   NAMED     PRESET CONSTANT
    +1010101010102622 DL________________________________0000000000000000 IS LONG    LOCAL    NAMED     PRESET CONSTANT
  40:99B2B3B0B9B9B2B6 message3__________________________0000000000000029 IS STRING  POOL     LOCATION  SET AT LINE 6
  42:9010101010103733 gn________________________________0000000000000008 IS LONG    POOL     LOCATION  SET AT LINE 3
  45:10101033BAB132B2 debug_____________________________00000000000001FF IS LONG    NUMBER   NAMED     PRESET CONSTANT
  46:2E0DBBF4019486F1 DIAGNOSTIC_OUTPUT_________________0000000000000002 IS LONG    NUMBER   NAMED     PRESET CONSTANT
  49:9A32B3B0B9B9B2B6 message4__________________________0000000000000032 IS STRING  POOL     LOCATION  SET AT LINE 7
    +D53486048564347D DEBUG_GRS________________________0000000000001000 IS LONG    NUMBER   NAMED     PRESET CONSTANT
  56:9010101035B2B2B9 seek______________________________0000000000000003 IS LONG    NUMBER   NAMED     PRESET CONSTANT
  60:101010101010 20A9 RA_______________________________0000000000000008 IS LONG    LOCAL    NAMED     PRESET CONSTANT
  62:90101010101033 18 1f_______________________________0000000000000000 IS LABEL   CODE     LOCATION
  63:90101010101033 19 3f_______________________________0000000000000000 IS LABEL   CODE     LOCATION
  64:901010101010331A 5f________________________________0000000000000000 IS LABEL   CODE     LOCATION
  65:901010101010331B 7f________________________________0000000000000000 IS LABEL   CODE     LOCATION
  66:901010101010331C 9f________________________________0000000000000000 IS LABEL   CODE     LOCATION
  69:542456048564 44FE DEBUG_DATA_____________________0000000000000800 IS LONG    NUMBER   NAMED     PRESET CONSTANT
  71:901010101035B1B0 ack_______________________________0000000000000000 IS ROUTINE EXTERNAL LOCATION  REF AT LINE 13
  79:D534C6048564347D DEBUG_KRS_________________________0000000000004000 IS LONG    NUMBER   NAMED     PRESET CONSTANT
  80:90101032BA34B93B write_____________________________0000000000000005 IS LONG    NUMBER   NAMED     PRESET CONSTANT
  83:16A7375546077B0C Put_String________________________0000000000000000 IS ROUTINE EXTERNAL LOCATION  REF AT LINE 12
    +9010103A30B2B931 creat_____________________________0000000000000001 IS LONG    NUMBER   NAMED     PRESET CONSTANT
  84:10323639AFBA3AB8 put_sld___________________________0000000000000101 IS LONG    NUMBER   NAMED     PRESET CONSTANT
  86:D53456048564347D DEBUG_DRS_________________________0000000000008000 IS LONG    NUMBER   NAMED     PRESET CONSTANT
  87:C4554E3EED0B39F4 STANDARD_INPUT____________________0000000000000000 IS LONG    NUMBER   NAMED     PRESET CONSTANT
  91:FD0633E554E86E6F calls_for_n_equal_10______________0000000000000000 IS LONG    POOL     LOCATION  SET AT LINE 2
  93:1039BA372FBA3AB8 put_nts___________________________0000000000000100 IS LONG    NUMBER   NAMED     PRESET CONSTANT
  95:90101032B9B7B631 close_____________________________0000000000000006 IS LONG    NUMBER   NAMED     PRESET CONSTANT
    +901010103732B837 open______________________________0000000000000002 IS LONG    NUMBER   NAMED     PRESET CONSTANT
  97:90323639AFBA32B3 get_sld___________________________0000000000000102 IS LONG    NUMBER   NAMED     PRESET CONSTANT
  99:901010103A34BC32 exit______________________________0000000000000000 IS LONG    NUMBER   NAMED     PRESET CONSTANT
 102:10101010101031 18 0b_______________________________0000000000000000 IS LABEL   CODE     LOCATION
 103:10101010101031 19 2b_______________________________0000000000000000 IS LABEL   CODE     LOCATION
 104:101010101010311A 4b________________________________0000000000000000 IS LABEL   CODE     LOCATION
    +1034363AAFBA3AB8 put_ulh__________________________0000000000000103 IS LONG    NUMBER   NAMED     PRESET CONSTANT
 105:101010101010311B 6b________________________________0000000000000000 IS LABEL   CODE     LOCATION
 106:101010101010311C 8b________________________________0000000000000000 IS LABEL   CODE     LOCATION
 111:C46F4EBC6F0EBCF4 STANDARD_OUTPUT__________________0000000000000001 IS LONG    NUMBER   NAMED     PRESET CONSTANT
 117:9034363AAFBA32B3 get_ulh__________________________0000000000000104 IS LONG    NUMBER   NAMED     PRESET CONSTANT
 119:1010101101029A7AA TOS_____________________________FFFFFFFFFFFFFFF8 IS LONG    LOCAL    NAMED     PRESET CONSTANT
   ... end of routine main
   Start of routine ack ...
==========================
The LOCAL symbol table for ack has 3 item(s) in 67 buckets, of which 3 are occupied.
==========================
  15:673667B6F706F62E m_nonzero_________________________00000000000000A0 IS LABEL   CODE     LOCATION  SET AT LINE 62
  37:773667B6F706F62E n_nonzero_________________________00000000000000B8 IS LABEL   CODE     LOCATION  SET AT LINE 69
  51:90101010101010 36 m________________________________FFFFFFFFFFFFFFF0 IS LONG    LOCAL    LOCATION  SET AT LINE 55
   ... end of routine ack
... 'ackw.a' was successfully assembled.

Assembly succeeded: no error was detected.
```

where we can see, e.g., that `message4` is a `STRING` constant, has offset `032` within the pool section, and was declared at line 7 of `ackw.a`; that `m_nonzero` is a label in the routine `ack`, has offset `0A0` in its code section, and was declared at line 62; and that `Put_String` is external, and was first referred to at line 12.

Assembly with the `-d` option may further list the tables used to recognise the instruction mnemonics and the reserved words of the A language. These are intended for debugging and optimising the assembler; they are of little interest to the A programmer.

L

THE RELOCATING LOADER

L1 PROGRAM LOADING

A simple relocating LOADER is provided that copies a program from one or more files into RAM. It reads files written in the format known as L. A compiler or assembler may output an object program in L format. All L program filenames have extension "`.l`". Progress messages from the loader are written to the log file `e_log.txt`.

L2 RELOCATION

The loader generates an executable object program that consists of five SEGMENTS, each of which has its natural length rounded up to, and is aligned on, a multiple of the (implementation-defined) PAGESIZE. These segments are:

- The (constant) POOL segment, an area starting at address 0, of length L_p, and extending towards higher addresses;
- The (executable) CODE segment, an area starting at address L_p, and of length L_c, and extending towards higher addresses;
- The (variable) HEAP segment, an area starting at address L_p+L_c, of dynamic length, and extending towards higher addresses;
- The (variable) DATA segment, an area starting at address RAMSIZE-L_d, and of length L_d, and extending towards higher addresses;
- The (variable) STACK segment, an area starting immediately beneath the data segment, of dynamic length, and extending downwards towards lower addresses.

The heap and stack segments are created at run-time by explicit program action. Once their base addresses are determined, the loader has nothing further to do with them. No distinction is made between statically initialized and uninitialized variable data areas, so far as their placement in the data segment is concerned. If the K operating system supports different access permissions for different areas, the pool segment is read-only, the code segment is read-only and executable, and the heap, data, and stack segments are readable and writable.

The pool, code and data segments are created at load-time by concatenating the pool, code and data SECTIONS of object code modules. These sections are created by the source-program assembler or compiler, and their components are identified as such in the L program. The loader takes two passes through the object code modules presented to it, the aim of the first (very simple) pass being to determine the sizes of these sections, severally and in aggregate. Each object module must contain a line that begins with an asterisk, followed by three hexadecimal numbers that are the (natural, i.e. not rounded up to PAGESIZE) sizes of the module's pool, code, and data sections, in that order.

The line giving the section sizes may appear at any point in the object code for a module, even as late as the last line; but loading will be slightly more efficient if it occurs earlier (ideally, as the first line).

From these values the loader calculates the lengths, and hence the starting addresses, of the program's segments. These serve as initial relocation bases for module sections as they are read into RAM during the second pass. It sets four RELATIVIZERS (see §L5), namely `_POOL_SEGMENT`, `_CODE_SEGMENT`, `_HEAP_SEGMENT`, and `_DATA_SEGMENT`, to the base addresses of the corresponding segments. The relativizers `_POOL_SEG_SIZE`, `_CODE_SEG_SIZE`, and `_DATA_SEG_SIZE`, are set to the loaded sizes of the corresponding segments. The relativizer `_HEAP_MAX_SIZE` is set to the maximum size of the heap (i.e. to the difference between the end of the code segment and the start of the data segment; this area in practice must be divided between the heap and the stack). The relativizer `_REAL_RAM_SIZE` is set to the implemented size of the M machine's RAM storage. All of these values are useful in addressing program components, and setting up execution state, at run-time.

As the second pass starts loading each module, it sets three relativizers, `_POOL`, `_CODE`, and `_DATA`, to the start addresses of this module's three sections. These values are useful in relocating module components, and are incremented by the lengths of the corresponding sections at the end of the module, ready for loading the next.

L3 THE L LANGUAGE FORMAT

L programs consist of a series of lines, each of which begins with a FLAG, which consist of either one or two characters. The rest of the line is processed in a way that depends on the flag. Some lines give data to be copied into RAM (which may be assumed to be initially zeroized). Others give directions to the loader (for which, see §4). All characters after an unquoted vertical bar ("|") are ignored, and may be used for end-of-line commentary. Lines that begin with a non-flag character are not permitted. Each letter flag is equally valid in upper case and in lower case.

L4 LOADER FLAG SPECIFICATIONS

Flag	Effect	Parameter(s)	Examples
space *tab* or I	load hexadecimal **i**nteger or **i**nstruction into next word	**exactly** 8 hexadecimal digits, underscores allowed, with optional relativizing	I 0000_0_0_F0 0000_0_3_54;END_PT I 0000_0_E_DD,POSN
"	load a character string into successive bytes	one or more CHARACTER IMAGES terminated by "	"FILE_NAME\0"
FD	load a f.p. **D**-type value into next longword	double-precision floating point number	FD 3.141591415914159E200
FE	load a f.p. **E**-type value into next word	single-precision floating point number	FE 3.141591
UB	load **u**nsigned value into next **b**yte	integer in hexadecimal, or a character image in quotes	UB 10 UB '!'
UH	load **u**nsigned value into next **h**alfword	integer in hexadecimal, or two character images in quotes	UH FFFF UH '\R\N'
UL	load **u**nsigned **l**ong integer into next **l**ongword	integer in hexadecimal, with optional relativizing	UL 0 UL FFFFFFFFFFFF + N
UW	load **u**nsigned integer into next **w**ord	integer in hexadecimal, with optional relativizing	UW FFFF > SHIFT UW 0+NEW_VAL

I

The I flag is used to load an instruction word, or other data word, into the next word of RAM. The word to be loaded must be given as exactly eight hexadecimal digits, embedded underscores being permitted for readability. This is the recommended way of entering instruction words. The *tab* character or a space may be used as a less obtrusive alternative to the I.

THE " FLAG

This flag is used to load a character string into successive bytes. The string contains a series of CHARACTER IMAGES, each of which is either a Latin-1 character, or one of the following ESCAPE SEQUENCES, provided to enable certain non-graphic characters to be input conveniently:

- \A the Audible Alarm (BEL) character (code 16#07#)
- \B the BS (Backspace) character (code 16#08#)
- \D the DEL character (code 16#7F#)
- \F the FF character (Form Feed, end-of-page, code 16#0C#)
- \H the HT character (Horizontal Tab, code 16#09#)
- \N the NL/LF character (New Line/Line Feed, end-of-line in K, code 16#0A#)
- \R the CR character (Carriage Return, code 16#0D#)
- \V the VT character (Vertical Tab, code 16#0B#)
- \0 the NUL character (Null, used as a file-name terminator in K, code 0)

For any other character (usefully including the quote character and the backslash character), preceding it by a backslash has the effect of including it in the string but ignoring its preceding backslash. So \" leaves a quote in the string, and \\ leaves a single backslash in the string.

The A, B, D, F, H, N, R and V escapes are also recognized as such in lower case.

FE FD

These flags are used to load a FLOATING POINT VALUE, into the next (long)word of RAM. The datum to be loaded may be given in any valid Ada format for a floating point number.

UB

The UB flag is used to load an 8-bit integer, given as one or more hexadecimal digits. Alternatively, the value can be given as a character image, enclosed consistently in either apostrophes (') or quotes (").

UH
The UH flag is used to load a 16-bit integer, given as one or more hexadecimal digits. Alternatively, the value can be given as a two character images, enclosed consistently in either apostrophes (') or quotes (").

UW
The UW flag is used to load a 32-bit integer, given as one or more hexadecimal digits.

UL
The UL flag is used to load a 64-bit integer, given as one or more hexadecimal digits.

L5 RELATIVIZERS

RELATIVIZERS are **load-time variables** that can be set to unsigned integer longword values. A relativizer is identified by an identifier, upper-case and lower-case forms of the same letter being **distinct** and underscores significant. Their most important use is to act as bases for relocating code and data areas in appropriate relative positions, and for fixing up instructions that refer to those positions. However, simple facilities are provided for arithmetic and bit-manipulation using relativizers. These may be useful (e.g.) for computing the size of a data structure given its starting address, number of components, and component size.

DIGIT RELATIVIZERS, i.e. forward and backward references to local labels identified by a single digit, are provided in the manner popularized by Knuth's (M)MIX assemblers. A digit relativizer is identified by a single decimal digit *d*, is forward referenced using the form *d***f** or *d***F**, and is backward referenced using the form *d***b** or *d***B**. These forms refer to the textually nearest definition of the local relativizer *d* in the specified direction.

An instruction or (long)word integer value to be loaded may be modified at load time by a value derived from a relativizer. This is especially useful when loading instruction words, but may also be applied to data integers, as well as to the parameters of the =, ?, L, and E flags.

Several kinds of relativizing are provided to meet a variety of needs. Comprehensive facilities are provided for address-formation and for arithmetic, logical and shift operations.

Form	Effect	Value generated
v . *c*	lower-primary relativizing: set N_p in primary format order	$v + c * 2^{16}$
v $ *c*	self-relative relativizing: set N_p to the IAR-relative offset of *v*	$v + (c - IAR) * 2^{16}$
v : *c*	upper-primary relativizing: set N_p in primary format order	*v* + (*c* **and** FFFF0000)
v ; *c*	lower-secondary relativizing: set N_s in secondary format order	$v + c * 2^{20}$
v , *c*	lower- tertiary relativizing: set N_t in tertiary format order	$v + c * 2^{24}$
v + *c*	full 64-bit longword addition	*v* + *c*
v – *c*	full 64-bit longword subtraction	*v* - *c*
v * *c*	full 64-bit longword multiplication	*v* * *c*
v / *c*	full 64-bit longword division	*v* / *c*
v % *c*	full 64-bit longword modulus remainder	*v* **mod** *c*
v & *c*	full 64-bit longword bitwise and	*v* **and** *c*
v ! *c*	full 64-bit longword bitwise inclusive or	*v* **or** *c*
v # *c*	full 64-bit longword bitwise exclusive or	*v* **xor** *c*
v < *c*	full 64-bit longword logical shift left	$v * 2^c$
v > *c*	full 64-bit longword logical shift right	$v / 2^c$

LOWER PRIMARY RELATIVIZING is intended for completing the N_p field of primary format instructions. It adds the **lower halfword** of the relativizer's value into the **upper halfword** of *v*. It is particularly useful for coding forward branches and other forward operand references.

SELF-RELATIVE RELATIVIZING is lower primary relativizing with a self-relative value. The value used to modify *v* is the value of *c* minus the value that IAR would contain at that point in the program. It adds the **lower halfword** of this relativized value into the **upper halfword** of *v*. It is particularly useful for writing position-independent code.

LOWER SECONDARY RELATIVIZING provides a similar service for secondary format instructions. It adds the **lower 12 bits** of the relativizer's value into the N_s field of the given value.

LOWER TERTIARY RELATIVIZING is intended for the N_t field of tertiary format instructions. It adds the **lower 8 bits** of the relativizer's value into the N_t field of the given value.

UPPER PRIMARY RELATIVIZING is intended for completing the N_p field of INSMHI instructions. It adds the **halfword** in **bits** 16:31 of the relativizer's value into the **upper halfword** of the given value.

THE = AND ? FLAGS

These flags are used to set or unset the values of relativizers. The = flag is used to set a relativizer. The ? flag unsets a specific relativizer. This allows already-set relativizers to be reused.

The form:

?*segment relativizer value*

is provided as a convenient and neater shorthand for the pair of lines:

?*relativizer*

=*segment relativizer value*

If a relativizer is used to modify any values loaded into RAM before the relativizer is set, the loader fixes up the contents of those locations when the relativizer is eventually set. A relativizer used to modify a parameter of the =, ?, L, and E flags must already be set at the point of use.

Flag	Effect	Parameter v	Examples
=**d*	set the digit relativizer *d* equal to the current load address (*d* is a decimal digit)	(none)	=*1 =*9
=*s c*	set the relativizer *c* equal to the current load address	(none)	=S end_pt = CurrAddr
=*s c v*	set the relativizer *c* equal to given value *v*	hexadecimal integer, with optional relativizing	=G start 0000_0280 =G Q 0ff0+start
?*s c*	unset the relativizer *c*	(none)	? CurrAddr
?*s c v*	reset the relativizer *c* to given value	hexadecimal integer, with optional relativizing	? Q 020+Q ?G Q 0ff0+start

A relativizer may be set either to a specified value, given as an unsigned integer in hexadecimal (with optional relativizing), or by default to the value of the current load address.

It is an error if a relativizer that has been used is not set before the end of the program. It is also an error to try to set a relativizer that was preset in the options prelude file.

The flag may optionally be followed by a segment marker, indicated by *s* in the table above. This provides the debugger with information as to the scope of the relativizer, and is one of the letters R, for a routine name; A, for an argument; G, for a global; P, for a pooled constant; L, for a local; or S, for a local static.

=*

The =* flag is used to set a digit relativizer *d*. It unsets any previous value of *d*b, sets a new value of *d*f to satisfy all outstanding forward references to *d*, unsets that value of *d*f, and then sets a new value of *d*b to satisfy any up-coming (backward) references to *d*.

L6 LOADING CONTROL, DEBUGGING FEATURE, AND COMMENTS

Flag	Effect	Parameter(s)	Examples
`A`	set the **a**lignment	1, 2, 4, 8, or 0 for *pagesize*	`A 2`
`B`	flag a **b**reakpoint	(none)	`B`
`E`	set the **e**ntry point	optional hexadecimal integer, optional relativizing	`E` `E 2C+GO_HERE`
`L`*s*	set the **l**oad address	optional hexadecimal integer, optional relativizing	`LC 0000_1000` `LP`
`W`	flag a **w**atchpoint	(none)	`W`
`|`	comment	(none)	`| A COMMENT`
`||`	logged comment	(none: comment is copied to `e_log.txt`)	`|| LOG THIS!`
`*`	define the section sizes	Three hexadecimal integers on one line, each preceded by a blank	`* 0:0 0:64 0:8`

A

This flag is used to set the ALIGNMENT, determining the way in which the loader rounds up the current load address. The load address starts at 0 by default, and is incremented by 1 for each byte or element of a string, by 2 for each halfword, by 4 for each word, and by 8 for each doubleword. The loader then skips to the next higher address which is a multiple of the alignment parameter, and resumes loading from that location. Notwithstanding the use of the `A` flag, the loader will never attempt to put a value into a location that it not suitably aligned for it. It ensures this by rounding up the load address to the next higher multiple of the item size (if necessary) before attempting to place the item in RAM. For example, if the alignment is set to 1, and the previous item left the load address at $\ldots2_{16}$, the loader will place a following word at $\ldots4_{16}$. The parameter may take only the values 1, 2, 4, or 0 for the implementation-defined *pagesize*. The default is 4.

B

This flag directs the loader to set a BREAKPOINT on the following location. When the program runs, this may indicate to a debugger that it should break off execution and interact with the user, if the flow of control reaches that location. (Setting a breakpoint does not affect the instruction loaded.)

E

The execution of the M program begins at the start of the code segment by default. The `E` flag sets the ENTRY POINT to the given address. Any address is given as a hexadecimal integer, with optional relativizing. However, the specified relativizer must already have been set.

L

This flag is used to set the LOAD ADDRESS. The L flag directs the loader to skip to a new load address, and resume loading from that location. The segment marker, *s* in the table above, specifies the segment into which the following material is to be loaded, and is one of the letters `C`, for the code segment; `G`, for the global data in the data segment; `P`, for the pooled constants segment; or `S`, for local static data in the data segment. Any address is given as a hexadecimal integer, with optional relativizing. However, the specified relativizer must already have been set. If no address is specified, loading resumes in the specified segment where it previously was suspended. A warning is issued if the new load address is less than the current load address (implying that a value previously loaded may now be overwritten).

W

This flag directs the loader to set a WATCHPOINT on the following location. When the program runs, this may indicate to a debugger that it should break off execution and interact with the user, if an instruction accesses that location as an operand. (Setting a watchpoint does not affect any value loaded.)

L7 EXAMPLE L PROGRAM FILE

See the corresponding A program in §A2. This is the file obtained by assembling `ackw.a` with default options:

```
* 00000050  000000D4  00000000
|L code for: 'ackw.a' at: 2017-02-12 00:46:01.09
=N gm 0000000000000003             |  1. *NUMBER LONG   gm = 3
LP
=P calls_for_n_equal_10 0000       |  2. *POOL   LONG   calls_for_n_equal_10 = 44780245
UL 0000000002AB4AD5
=P gn 0008                         |  3. *POOL   LONG   gn = 10
UL 000000000000000A
=P message1 0010                   |  4. *POOL   STRING message1 = "Ackermann(3, \0"
"Ackermann(3, \0"
=P message2 0021                   |  5. *POOL   STRING message2 = ") = \0"
") = \0"
=P message3 0029                   |  6. *POOL   STRING message3 = ".\n\n\0"
".\n\n\0"
=P message4 0032                   |  7. *POOL   STRING message4 = " ns/call\n\n\0"
" ns/call\n\n\0"
RB main 0000                       |  9. *PROGRAM main
LC
E
                                   | 11. *EXTERNAL ROUTINE Put_Long
                                   | 12. *EXTERNAL ROUTINE Put_String
                                   | 13. *EXTERNAL ROUTINE ack
I FFF8_0_E_17                      | 16.    loduli   FPR   TOS
I 0000_E_E_0F                      | 17.    strul    FPR   DL[FPR]
I FFE8_0_D_17                      | 18.    loduli   SPR   -24
I FFF8_E_D_0F                      | 19.    strul    SPR   TOS[FPR]
LP
=P _pool_0048 0048
UL FFFFFFFFFFF00000
LC
I 0048_0_2_1F._POOL                | 20.    lodul    gr2   = #FFFFFFFFFFF00000
I 0004_2_C_C9                      | 21.    cpygs    SLT   gr2
I 0010_0_1_17._POOL                | 22.    loduli   gr1   message1
I 0000_0_A_B0.Put_String           | 23.    gosub    gra   Put_String
I 0008_0_1_1F._POOL                | 24.    lodul    gr1   gn
I 0000_0_A_B0.Put_Long             | 25.    gosub    gra   Put_Long
I 0021_0_1_17._POOL                | 26.    loduli   gr1   message2
I 0000_0_A_B0.Put_String           | 27.    gosub    gra   Put_String
I 0003_0_7_17                      | 28.    loduli   gr7   gm
I 0008_0_8_1F._POOL                | 29.    lodul    gr8   gn
I 0001_1_6_C9                      | 30.    cpysg    gr6   NSR
I 0000_0_E_B8.ack                  | 31.    call     FPR   ack
I 0001_1_5_C9                      | 32.    cpysg    gr5   NSR
I 0000_6_5_33                      | 33.    subsli   gr5   0[gr6]
I 0001_5_1_FA                      | 34.    loddi    dr1   0[gr5]
I 0000_0_2_1B._POOL                | 35.    lodsl    gr2   calls_for_n_equal_10
I 0001_2_2_FA                      | 36.    loddi    dr2   0[gr2]
I 0015_2_3_C8                      | 37.    divdr    dr3   dr1, dr2
I 0000_0_1_C9                      | 38.    cpygg    gr1   gr0
I 0000_0_A_B0.Put_Long             | 39.    gosub    gra   Put_Long
I 0029_0_1_17._POOL                | 40.    loduli   gr1   message3
I 0000_0_A_B0.Put_String           | 41.    gosub    gra   Put_String
I 0008_3_3_C8                      | 42.    rnddr    dr3   dr3
I 0000_3_1_C8                      | 43.    condr    gr1   dr3
I 0000_0_A_B0.Put_Long             | 44.    gosub    gra   Put_Long
I 0032_0_1_17._POOL                | 45.    loduli   gr1   message4
I 0000_0_A_B0.Put_String           | 46.    gosub    gra   Put_String
I 0000_0_0_FC                      | 47.    sysc           exit
RE main                            | 48. *END
RB ack 0080                        | 50. *ROUTINE ack
LC
|| the argument m is in gr7, n in gr8
?L m FFF0                          | 52. *LOCAL LONG m | n need not be stacked - tail recursion on n
LC
I 0000_0_0_FE                      | 54.    enter
I FFF0_D_0_FF                      | 55.    frame          m[SPR]
I 0000_F_7_B5$m_nonzero            | 56.    bgtz     gr7   $m_nonzero
I 0001_8_0_13                      | 57.    lodsli   gr0   1[gr8]        | return n+1 if m = 0
I 0000_0_0_F2                      | 58.    leave                         | 5 inst to here
?C m_nonzero 0094                  | 59. m_nonzero:
I 0000_F_8_B5$n_nonzero            | 60.    bgtz     gr8   $n_nonzero
I 0001_0_7_33                      | 61.    subsli   gr7   1
I 0001_0_8_13                      | 62.    lodsli   gr8   1
I 0080_0_E_B8._CODE                | 63.    call     FPR   ack
I FFF8_E_D_F5                      | 64.    trim     SPR   TOS[FPR]       | return Ack(m-1, 1), if n = 0
I 0000_0_0_F2                      | 65.    leave
```

```
?C n_nonzero 00AC                  | 66. n_nonzero:
I FFF0_E_7_0B                      | 67.    strsl    gr7   m
I 0001_0_8_33                      | 68.    subsli   gr8   1
I 0080_0_E_B8._CODE                | 69.    call     FPR   ack          | compute Ack(m, n-1)
I FFF8_E_D_F5                      | 70.    trim     SPR   TOS[FPR]
I FFF0_E_7_1B                      | 71.    lodsl    gr7   m
I 0001_0_7_33                      | 72.    subsli   gr7   1
I 0000_0_8_C9                      | 73.    cpygg    gr8   gr0
I 0080_0_E_B8._CODE                | 74.    call     FPR   ack
I FFF8_E_D_F5                      | 75.    trim     SPR   TOS[FPR]     | return Ack(m-1, Ack(m, n-1))
I 0000_0_0_F2                      | 76.    leave                       | 14 inst to here, mean 9.5
RE ack                             | 77. *END ack
|Finish of A file: 'ackw.a'
```

An instruction such as:

```
I 0003_0_7_17                      | 28.    loduli   gr7   gm
```

is complete: everything needed to formulate it was known to the assembler, including the value (3) of the operand `gm`, declared as such earlier in the module.

These instructions:

```
I 0008_0_1_1F._POOL                | 24.    lodul    gr1   gn
I 0000_0_A_B0.Put_Long             | 25.    gosub    gra   Put_Long
```

are incomplete: the operands `gn and Put_Long` are the, as yet unknown, addresses of locations.

`gn` is a longword operand at offset 8 within this module's pool section; what remains to be determined is the start of that section when placed within the whole program's pool segment. The assembler flags this up by appending the relativizer "`._POOL`" to the hexadecimal. When this module is relocated by the loader, the value it determines for `POOL` will be added to the N-part (8), to form a complete instruction.

`Put_Long` is not declared in this module, and the assembler flags this up by appending the relativizer "`.Put_Long`" to the hexadecimal. When the loader encounters the declaration of `Put_Long`, in a separate module, its then-known absolute address will be added to the N-part, to form a complete instruction.

E

THE EMULATOR

E1 THE EMULATOR OPTIONS FILE

The emulator has a number of diagnostic facilities that are controlled by user-settable parameters. It precedes loading by attempting to read values for these parameters from a file named `prelude.txt`. If this is unsuccessful, either for an individual flag or for the file as a whole, default settings prevail for the parameter(s) affected. After loading the program, it similarly reads `postlude.txt`, which can depend on relativizers set during the loading process. Both `prelude.txt` and `postlude.txt` consist of a series of lines in the same format as an L program file, but with a different selection of allowable flags.

In addition to the flags described below, the '=' flag may also be given in `prelude.txt`, to PRESET relativizers that are used (e.g.) to specify load addresses and entry point within a program file, or to adjust the parameters of the `C`, `P`, `R` and `T` flags. Relativizers used in parameters of the `C`, `P`, `R` and `T` flags must already be set. Relativizer values preset in `prelude.txt` cannot be changed unless they re explicitly unset first (in a program file).

E2 OPTIONS FLAG SPECIFICATIONS

Flag	Effect	Parameter(s)	Example
`C`	set the instruction tracing **c**ounts	two integers (decimal or hexadecimal), optional relativizing	`C 80 90` `C 0 #A0`
`D`	set the instruction stepping **d**elay	floating point number of seconds	`D 1.5`
`I`*t*	set the **i**nitial RAM printing options	two hexadecimal integers, optional relativizing	`IL 0+THERE A0+THERE IG`
`L`	set the execution time **l**imit	one integer (decimal or hexadecimal), optional relativizing	`L 5_000_000`
`M`	set the execution **m**ode	mode name	`M NORMAL_MODE`
`R`	set the instruction tracing address **r**ange	two **hexadecimal** addresses, optional relativizing	`R 0 0100+T`
`S`	set the execution time **s**lice	one integer (decimal or hexadecimal), optional relativizing	`S 5_000`
`T`*t*	set the **t**erminal RAM printing options	two hexadecimal integers, optional relativizing	`TWI 0+THERE A0+THERE PCFG`

M

This flag is used to set the execution MODE, specifying the level of debugging and the kind of tracing output that may be generated. The mode can be specified in upper case or in lower case. The alternatives fall into two independent groups, which are: `FASTEST_MODE`, `NORMAL_MODE`; and: `RUNNING_MODE`, `STEPPING_MODE`. The first group sets the level of diagnostic information; the second group sets the frequency at which the debugger interacts.

	FASTEST MODE	NORMAL MODE
RUNNING MODE	full speed with no debugging facilities	full speed; retrospective tracing
STEPPING MODE	rate given by the `X` flag; effect displayed for each instruction step	execute one instruction; effect displayed before pausing for the next instruction

The output provided at a breakpoint or on termination, in every mode except `RUNNING_MODE` with `NORMAL_MODE`, includes a RETROSPECTIVE TRACE of the branches executed since the last such output. The trace includes the address of the traced instruction itself, and that of the destination of the branch. If an instruction goes repeatedly to same destination, with no other branch intervening, this is indicated. The branches are listed in order, starting with the most recently executed.

The output provided in `STEPPING_MODE` includes the address of the instruction itself, and the end result left in any updated register or RAM location.

While tracing is suspended due to the limits imposed by the `R` and `C` flags, the effective mode is equivalent to `FASTEST_MODE` with `RUNNING_MODE`.

The emulator reflects an implementation of the M architecture using several internal registers (see §1), and these may appear in traces.

C

This flag is used to set two COUNT values, say l and h, that determine whether tracing output may be generated. No tracing output is given for any instruction execution unless $l \le i \le h$ is satisfied, where i is the current value of the Instruction Count Register. With suitable l and h values, tracing can confined to a set time during execution (for example, the last few instruction executions before the program fails). Their defaults are 0 and 20 respectively. See also the `R` flag.

D

This flag is used to set an INTER-INSTRUCTION DELAY, say d ($0.0 \le d \le 15.0$) which is used by the debugger to pace the production of tracing output when running in `STEPPING_MODE`. The delay, representing a time in seconds, may be given in any valid Ada format for a floating point number.

I

Before execution begins, i.e. INITIALLY, print to `e_log.txt` an area of RAM between two addresses, given as hexadecimal integers with optional relativizing. Used in combination with postmortem diagnostic printing, this allows for "before and after" comparison of storage.

RAM in the specified range may be output (a) as unsigned hexadecimal integers in byte, word and longword forms; (b) as signed decimal integers, in word and longword forms; (c) as Latin-1 characters (NUL is shown as the masculine ordinal indicator 'º', and any other non-printing character as the inverted question mark '¿', so that they are clearly distinguished from spaces); (d) as floating point numbers in single and double precision forms; and (e) as instructions disassembled in an approximation to A format. The type marker, t in the table above, is a string of one or more of the letters `BDEHLSW`: denoting interpretation of the designated area as **b**ytes, **D**-format floating point, **E**-format floating point, **h**alfwords, **l**ongwords, **s**trings, and **w**ords respectively. The further letter `G` requests printing of the **g**lobal data area in a fixed format. If the letter `X` is included, all previously-set requests with address ranges that are included in the present range, and which have type markers included in the present set of type markers, are removed from the list of requests.

L

This flag is used to set a value, say t, that specifies an execution TIME LIMIT. This determines how long the M program is allowed to execute before being terminated. The limit is specified in instruction executions rather than seconds, so the program is terminated if `ICR` $> t$ at the end of any instruction execution. The default gives about 10 seconds of CPU time.

R

This flag is used to set two values, say a and b, that delimit the RANGE of instructions for which tracing output may be generated. No tracing output is given for any instruction unless the condition $a \le i \le b$ is satisfied, where i is the current value of `IAR` (before the instruction is fetched). With suitable a and b values, instruction tracing can confined to the sequence of instructions that you are currently debugging. The values a and b must be given as hexadecimal integers, with optional relativizing. The defaults are 0 and the highest RAM address, respectively. See also the `C` flag.

S

This flag is used to set a value, say t, that specifies an execution TIME SLICE. This determines how long the M program is allowed to execute before being pre-empted to allow user intervention or multi-task context switching (the latter only when the implementation of K supports it). The limit is specified in instruction executions, so the program is pre-empted after every t instruction executions. The default is a small fraction of the time limit. t is automatically bounded above, to preserve responsiveness should an infeasibly large value be given.

T

When execution TERMINATES, print to `e_log.txt` an area of RAM between two addresses, given as hexadecimal integers with optional relativizing. Used in combination with initial diagnostic printing, this allows for "before and after" comparison of storage. The type marker, t in the table above, denotes the same options as for the `I`, pre-run request, with the addition of the letters `C` and `F`: denoting the **c**ontext (i.e. the instructions around the address in `IAR`), and the stack **f**rame given by the addresses in `FPR` and `SPR`). The printed execution context may indicate a particular instruction as the 'current' instruction. If execution terminated during the execution of an instruction (e.g. as the result of a machine trap), then that is the current instruction; but if execution terminated after completion of an instruction (e.g. as the result of exceeding the time limit, or at the user's request) the current instruction is that which would next have been executed.

E3 Running the emulator from the command line

The emulator can be invoked from the command line, thus:

e [**-u** | **-d** | **-l***f* | **-e***m* | *file_argument*] …

where *file_argument* names a program file.

Logging flags invoke options that apply to program files named to their right in the argument list; other flags apply to the whole list.

Examples:

```
e
e  -u
e  -ls subr1.l –est subr2.l main.l
e  -u program.l
```

A program file parameter should be the name of a file with '`.l`' extension, containing code in L format. Each such file is loaded in turn. (relativizer settings made in one file are carried over to the next in the list.) If there is no program file parameter, `program.l` is used by default. Consequently the second and fourth examples behave in exactly the same way.

As a concession to users of command-line completion, it is permissible to omit the final '`.l`' from a program file name, or even just the final '`l`' so that the parameter ends with '`.`'; the emulator will complete the name automatically.

The available **e**xecution mode flags *m* are one or two of the following:

`f`: **f**astest execution mode;

`n`: **n**ormal execution mode;

`r`: **r**unning execution mode;

`s`: **s**tepping execution mode.

Logging messages that record the progress of the loading, relocation and execution, and details of any errors encountered, are written to the file `e_log.txt`. The **l**evel of detail to be recorded may be controlled by the **-l***f* parameter, as follows:

`-ls`: **s**uppresses all non-diagnostic logging output

`-ln`: provides **n**ormal logging output

`-le`: provides **e**xtensive output, including the contents of the local or global symbol tables, as appropriate.

The `-u` flag indicates that the run is **u**nattended (e.g. part of a background job with no user present) and forces non-interactive execution, regardless of the mode specified in any options file. Specifically, `STEPPING_MODE` is ignored and the inter-instruction delay is set to 0.0. When `-u` is absent, `STEPPING_MODE` is honoured and the requested inter-instruction delay is used.

The **-d** flag engages a mode in which optional internal consistency checks may be made, and internal tables that might be useful **d**iagnostics for the implementor of the emulator may be output. This is intended primarily for diagnosing errors in **e** itself. Such output will be generated only if **e** was compiled by a debugging build.

In `NORMAL_MODE` with `STEPPING_MODE` the emulator uses console-window text I/O to display diagnostic output and to take in debugger commands. At each traced instruction or breakpoint, a prompt asks the user how to continue. The user replies with a single letter, selecting one of the following:

`f`: execution proceeds in fastest mode;

`n`: execution proceeds in normal mode;

`r`: execution proceeds in running mode;

`s`: execution proceeds in stepping mode.

If the response is `r` or `s`, the debugger prompts a second time, to allow the diagnostic level to be varied as well. If the response is a carriage return execution resumes in the (now) current modes.

E4 Example of running the emulator

Here is a run of the example program in §L2:

```
/Users/wf/Mekhos/Testing/ack: e ackw puts putl
Loader: Program loading begins.

_____________________________________
Sizes from 'ackw.l':
Size of pool section = 0000000000000050
Size of code section = 00000000000000E0
Size of data section = 0000000000000000

_____________________________________
Sizes from 'puts.l':
Size of pool section = 0000000000000000
Size of code section = 0000000000000010
Size of data section = 0000000000000000

_____________________________________
Sizes from 'putl.l':
Size of pool section = 0000000000000000
Size of code section = 0000000000000014
Size of data section = 0000000000000000

_____________________________________________________________________________
Storage layout for the loaded program is as follows:

Pool segment base = 0000000000000100
Pool segment size = 0000000000000100

Code segment base = 0000000000000200
Code segment size = 0000000000000200

Heap segment base = 0000000000000400
Heap segment size = 000000000007FC00

Data segment base = FFFFFFFFFFFFFF00
Data segment size = 0000000000000100

Stack upper bound = FFFFFFFFFFFFFE00
Stack lower bound = FFFFFFFFFFF80000

RAM  storage size = 0000000000100000
Program load size = 0000000000000400

_____________________________________________________________________________
_____________________________________
Loading the sections of 'ackw.l':
Pool section base = 0000000000000100
Code section base = 0000000000000200
Data section base = FFFFFFFFFFFFFF00
gm_____________________________0000000000000003 is a number, set at line 3
calls_for_n_equal_10___________0000000000000100 is a pool address, set at line 5
gn_____________________________0000000000000108 is a pool address, set at line 7
message1_______________________0000000000000110 is a pool address, set at line 9
message2_______________________0000000000000121 is a pool address, set at line 11
message3_______________________0000000000000129 is a pool address, set at line 13
message4_______________________0000000000000132 is a pool address, set at line 15

ROUTINE   main
Loads at  0000000000000200
ENTRY AT  0000000000000200
_pool_0048_____________________0000000000000148 is a pool address, set at line 28

ROUTINE   ack
Loads at  000000000000028C
m______________________________000000000000FFF0 is a frame local offset, set at line 66
m_nonzero______________________00000000000002A0 is a code address, set at line 73
n_nonzero______________________00000000000002B8 is a code address, set at line 80
L module 'ackw.l' successfully loaded; ending at line  92.
_____________________________________
Loading the sections of 'puts.l':
Pool section base = 0000000000000150
Code section base = 00000000000002E0
Data section base = FFFFFFFFFFFFFF00

ROUTINE   Put_String
Loads at  00000000000002E0
L module 'puts.l' successfully loaded; ending at line  10.
_____________________________________
Loading the sections of 'putl.l':
Pool section base = 0000000000000150
Code section base = 00000000000002F0
Data section base = FFFFFFFFFFFFFF00

ROUTINE   Put_Long
Loads at  00000000000002F0
L module 'putl.l' successfully loaded; ending at line  11.
```

We can see, e.g., that `message4`, which has offset `0:32` within its module's pool section, has the absolute address `0:132` within the program's complete, relocated, virtual address space, because the module's pool section starts at location `0:100` (previous modules not having allocated any pool space). Similarly, `m_nonzero`, which has offset `0:94` within its module's code section, has the absolute address `0:294` within the program's complete, relocated, virtual address space, because the module's code section starts at location `0:200`.

The main program's entry point and address are clearly marked. The emulator uses this information to set the Instruction Address Register to its initial value for the run. The object program having been loaded, the emulator executes it. I have coloured the program's output red, to make it stand out clearly from the emulator's chatter:

```
Emulator: Running in normal mode (without diagnostics).
```

```
Ackermann(3, 10) = 8189.

82 ns/call
```

```
================================================================================
Current state, with [IAR] = 0000028C

Internal Registers.
[CIR] = #0000_0_0_FC      = sysc   exit
[MAR] = #0000000000000000 =                    0
[MBR] = #41855A56A8000000 =   4721279112600092672 =  4.47802450000000E+07

Special Registers.
[CCR]= #2017100414540302 =   2312334543385002754, i.e., Wed, 2017-Oct-04 at 14:54:03
[NSR]= #00000000DA721A98 =            3664911000, for about 9 ns/M-instruction
[ICR]= #000000000194F572C =             424630060, for about 115 M MIPS
[TRA]..[BTR] are all zero.
[SLT]= #FFFFFFFFFFF80008 =               -524280
[SPC]= #0000000000000000 =                     0
[FPC]= #FFFFFFFFFFFFFF98 =                  -104
[IAC]= #00000000000002D8 =                   728

General Registers.
[gr0]= #000000000000000A =                    10
[gr1]= #0000000000000001 =                     1
[gr2]= #0000000000000132 =                   306
[gr3]= #0000000000000001 =                     1
[gr4]= #0000000000000000 =                     0
[gr5]= #00000000DA6A0830 =            3664382000
[gr6]= #000000000000A028 =                 41000
[gr7]= #0000000000000000 =                     0
[gr8]= #0000000000001FFC =                  8188
[gr9]= #0000000000000000 =                     0
[gra]= #0000000000000288 =                   648
[grb]= #000000000194F56EC =             424629996
[EPR]= #FFFFFFFFFFFFFFD8 =                   -40
[SPR]= #FFFFFFFFFFFFFFC0 =                   -64
[FPR]= #FFFFFFFFFFFFFFF8 =                    -8
[IAR]= #000000000000028C =                   652

Floating Point Registers.
[dr0]= #0000000000000000 =  0.00000000000000E+00
[dr1]= #41EB4D4106000000 =  3.66438200000000E+09
[dr2]= #41855A56A8000000 =  4.47802450000000E+07
[dr3]= #4054800000000000 =  8.20000000000000E+01
[dr4]..[drf] are all zero.

Kernel Registers.
[ESF]= #0000000000000001 =                     1, i.e., user state
[IDN]..[krf] are all zero.
```

```
There is no retrospective trace.
```

```
Current execution context.

RAM as (32-bit) instructions:
main:        lodshi FPR TOS                        [000002C4] = trim   SPR TOS[FPR]
[00000204] = strul  FPR (RA - 8)[FPR]              [000002C8] = lodsl  gr7 (TOS - 8)[FPR]
[00000208] = lodshi SPR -32                        [000002CC] = subsli gr7 1
[0000020C] = strul  SPR TOS[FPR]                   [000002D0] = cpygg  gr8 gr0
[00000210] = lodul  gr2 _pool_0048                 [000002D4] = call   FPR ack
[00000214] = cpygs  SLT gr2                        [000002D8] = trim   SPR TOS[FPR]
[00000218] = loduli gr1 message1                   [000002DC] = leave
[0000021C] = gosub  gra Put_String                 Put_String:  cpygg  gr2 gr1
[00000220] = lodul  gr1 gn                         [000002E4] = lodsli gr1 1
[00000224] = gosub  gra Put_Long                   [000002E8] = sysc   put_nts
[00000228] = loduli gr1 message2                   [000002EC] = goback 0[gra]
[0000022C] = gosub  gra Put_String                 Put_Long:    lodsli gr3 1
[00000230] = loduli gr7 3                          [000002F4] = cpygg  gr2 gr1
[00000234] = lodul  gr8 gn                         [000002F8] = lodsli gr1 1
[00000238] = cpysg  gr6 NSR                        [000002FC] = sysc   put_sld
[0000023C] = cpysg  gra ICR                        [00000300] = goback 0[gra]
[00000240] = call   FPR ack                        [00000304] = 0000_0_0_00
[00000244] = cpysg  grb ICR                        [00000308] = 0000_0_0_00
[00000248] = cpysg  gr5 NSR                        [0000030C] = 0000_0_0_00
[0000024C] = subsli gr5 0[gr6]                     [00000310] = 0000_0_0_00
[00000250] = loddi  dr1 0[gr5]                     [00000314] = 0000_0_0_00
[00000254] = subuli grb 0[gra]                     [00000318] = 0000_0_0_00
[00000258] = lodsl  gr2 calls_for_n_equal_10       [0000031C] = 0000_0_0_00
[0000025C] = loddi  dr2 0[gr2]                     [00000320] = 0000_0_0_00
[00000260] = divdr  dr3 dr1, dr2                   [00000324] = 0000_0_0_00
[00000264] = cpygg  gr1 gr0                        [00000328] = 0000_0_0_00
[00000268] = gosub  gra Put_Long                   [0000032C] = 0000_0_0_00
[0000026C] = loduli gr1 message3                   [00000330] = 0000_0_0_00
[00000270] = gosub  gra Put_String                 [00000334] = 0000_0_0_00
[00000274] = rnddr  dr3 dr3                        [00000338] = 0000_0_0_00
[00000278] = condr  gr1 dr3                        [0000033C] = 0000_0_0_00
[0000027C] = gosub  gra Put_Long                   [00000340] = 0000_0_0_00
[00000280] = loduli gr1 message4                   [00000344] = 0000_0_0_00
[00000284] = gosub  gra Put_String                 [00000348] = 0000_0_0_00
_____________________________________
[00000288] = sysc   exit                           [0000034C] = 0000_0_0_00
_____________________________________
ack:         enter                                 [00000350] = 0000_0_0_00
[00000290] = frame  (TOS - 8)[SPR]                 [00000354] = 0000_0_0_00
[00000294] = bgtz   gr7 $m_nonzero                 [00000358] = 0000_0_0_00
[00000298] = lodsli gr0 1[gr8]                     [0000035C] = 0000_0_0_00
[0000029C] = leave                                 [00000360] = 0000_0_0_00
m_nonzero:   bgtz   gr8 $n_nonzero                 [00000364] = 0000_0_0_00
[000002A4] = subsli gr7 1                          [00000368] = 0000_0_0_00
[000002A8] = lodsli gr8 1                          [0000036C] = 0000_0_0_00
[000002AC] = call   FPR ack                        [00000370] = 0000_0_0_00
[000002B0] = trim   SPR TOS[FPR]                   [00000374] = 0000_0_0_00
[000002B4] = leave                                 [00000378] = 0000_0_0_00
n_nonzero:   strsl  gr7 (TOS - 8)[FPR]             [0000037C] = 0000_0_0_00
[000002BC] = subsli gr8 1                          [00000380] = 0000_0_0_00
[000002C0] = call   FPR ack                        [00000384] = 0000_0_0_00
_____________________________________________________________________________

End of Run.
=============================================================================
Emulator: Run terminated normally.
```

One of the diagnostics is a dis-assembled display of the program instructions in the vicinity of the point of failure, with the last instruction to be executed clearly marked. In this case it is the system call instruction at location `027C`. In the following output, the M program was executed with run-time diagnostic facilities engaged. These are of two kinds:

- a postmortem display of an execution trace showing the flow of control up to termination, and
- step-by-step execution under interactive control.

Here is the execution trace for the program above, terminated by the (operator-enforced) end of the run. The number at the end of each line shows how many times that jump was taken consecutively before a different jump took place.

```
Retrospective branch trace.

Last traced branch:
[0000027C] = sysc   exit                                                    * 1
[000002E0] = goback 0[gra]                  [0000027C] = sysc   exit        * 1
[000002DC] = sysc   put_nts                 [ KERNEL ]                      * 1
[00000278] = gosub  gra Put_String          Put_String:  cpygg  gr2 gr1     * 1
[000002F4] = goback 0[gra]                  [00000274] = loduli gr1 message4 * 1
[000002F0] = sysc   put_sld                 [ KERNEL ]                      * 1
[00000270] = gosub  gra Put_Long            Put_Long:    lodsli gr3 1       * 1
[000002E0] = goback 0[gra]                  [00000268] = rnddr  dr3 dr3     * 1
[000002DC] = sysc   put_nts                 [ KERNEL ]                      * 1
[00000264] = gosub  gra Put_String          Put_String:  cpygg  gr2 gr1     * 1
[000002F4] = goback 0[gra]                  [00000260] = loduli gr1 message3 * 1
[000002F0] = sysc   put_sld                 [ KERNEL ]                      * 1
[0000025C] = gosub  gra Put_Long            Put_Long:    lodsli gr3 1       * 1
[000002D0] = leave                          [00000240] = cpysg  gr5 NSR     * 1
[000002D0] = leave                          [000002CC] = trim   SPR TOS[FPR] * 2
[00000290] = leave                          [000002CC] = trim   SPR TOS[FPR] * 1
[000002C8] = call   FPR ack                 ack:         enter              * 1
...
After: earlier instructions, whose tracing is now lost.
```

When run in single-stepping mode, the emulator displays the registers and/or storage locations that have been affected by the execution of the instruction, e.g.:

```
At 000000000000023C, CIR = #0280_0_E_B8 = call   FPR ack
[IAR]-->     IAC = #0000000000000240 =                  576
[FPR]-->     FPC = #FFFFFFFFFFFFFFF8 =                   -8
(n+x)-->     IAR = #0000000000000280 =                  640

D: (r:unning | s:tepping) or (n:ormal | t:racing) or q:uit?

At 0000000000000280, CIR = #0000_0_0_FE = enter
@[MAR]<--   [IAC]= #000000000000240 =                   576
[SPR]-8-->   SPR = #FFFFFFFFFFFFFFE0 =                  -32
[SPR]-->     FPR = #FFFFFFFFFFFFFFE0 =                  -32
@[MAR]<--   [FPC]= #FFFFFFFFFFFFFFF8 =                   -8
[SPR]-8-->   SPR = #FFFFFFFFFFFFFFD8 =                  -40

D: (r:unning | s:tepping) or (n:ormal | t:racing) or q:uit?

At 0000000000000284, CIR = #FFF0_D_0_FF = frame  (TOS - 8)[SPR]
@[MAR]<--    MBR = #FFFFFFFFFFFFFFC8 =                  -56
[MBR]-->     SPR = #FFFFFFFFFFFFFFC8 =                  -56

D: (r:unning | s:tepping) or (n:ormal | t:racing) or q:uit?

At 0000000000000288, CIR = #0008_F_7_B5 = bgtz   gr7 $m_nonzero
(n+x)-->     IAR = #0000000000000294 =                  660

D: (r:unning | s:tepping) or (n:ormal | t:racing) or q:uit?

At 0000000000000294, CIR = #0014_F_8_B5 = bgtz   gr8 $n_nonzero
(n+x)-->     IAR = #00000000000002AC =                  684

D: (r:unning | s:tepping) or (n:ormal | t:racing) or q:uit?

At 00000000000002AC, CIR = #FFF0_E_7_0B = strsl  gr7 (TOS - 8)[FPR]
@[MAR]<--   [gr7]= #0000000000000003 =                    3

D: (r:unning | s:tepping) or (n:ormal | t:racing) or q:uit?

At 00000000000002B0, CIR = #0001_0_8_33 = subsli gr8 1
result-->    gr8 = #0000000000000009 =                    9
...
```

MEKHOS REFERENCE MANUAL

K

THE OPERATING SYSTEM KERNEL

K1, THE MEKHOS OPERATING SYSTEM

K is the kernel of the emulated Operating System for **mekhos**. K/1 provides a very simple subset of the system calls common to most implementations of UNIX.

K2 SYSTEM CALLS

K is invoked by a process from user state with the `sysc` instruction, which causes a trap to the kernel. The operand of **`sysc`** is given by the immediate-mode interpretation of its N and X fields, the result being the SYSCALL NUMBER, which identifies the operation. Parameters are passed to the kernel as if the `sysc` instruction invoked a function taking parameters in register(s) `GR1`, …, and delivering a result in `GR0`. String parameters, such as filenames, are in UNIX format (NUL terminated) and passed by reference.

On completion of the system call, a termination code is returned in `ENR`. This is zero if the system call was successful, in which case `GR0` contains the result. If the system call fails, execution is terminated and `ENR` is set to an implementation-defined K error code, in which case the content of `GR0` is undefined. In addition, a diagnostic message to identify the system call and its operands is written to the standard error output, if possible.

The following system calls are presently available in K/1:

K System Call	Syscall number
`f_nr := creat (file : String; mode : longword range 0..8#777#);`	1
`f_nr := open (file : String; mode : longword range 0..2);`	2
`offset := seek (f_nr, offset : longword; whence : longword range 0..2);`	3
`count := read (f_nr, buffer, count : longword);`	4
`count := write (f_nr, buffer, count : longword);`	5
`status := close (f_nr : longword);`	6
`count := put_nts (f_nr, buffer : longword);`	-1
`count := put_sld (f_nr, number, width : longword);`	-2
`count := get_sld (f_nr, buffer : longword);`	-3
`count := put_ulh (f_nr, number: longword);`	-4
`count := get_ulh (f_nr, buffer : longword);`	-5
`diagnose`	2^{10} .. 2^{16}-1
`exit (final_status : longword);`	0

The operands and effects of system calls 1 through 6 are defined as for the corresponding calls in the host's UNIX Operating System.

System calls -1 through -5, and the *`diagnose`* calls, though not standard UNIX features, provide convenient facilities in the context of **mekhos**.

- `put_nts` write the **n**ull-**t**erminated **s**tring addressed by the value of `buffer`
- `put_sld` write the **s**igned **l**ong **d**ecimal number, `number`, right adjusted in a field of given `width`
- `get_sld` read a **s**igned **l**ong **d**ecimal number into the 8 bytes addressed by the value of `buffer`
- `put_ulh` write the **u**nsigned **l**ong **h**exadecimal number, `number`
- `get_ulh` read an **u**nsigned **l**ong **h**exadecimal number into the 8 bytes addressed by the value of `buffer`

The assembler predefines identifiers for the SYSCALL NUMBERs, for use as N-part operands of the `sysc` instruction.

Syscall number	Pre-defined Identifier
-5	GET_ULH
-4	PUT_ULH
-3	GET_SLD
-2	PUT_SLD
-1	PUT_NTS
0	EXIT
1	CREAT
2	OPEN
3	SEEK
4	READ
5	WRITE
6	CLOSE

The *diagnose* calls output a selection of internal state information that might be of help in revealing bugs in an executing program. The operand is bit-significant: each bit selects some data to be displayed; any combination of these bits is also a valid operand, with DEBUG_ALL requesting everything:

Option bit	Pre-defined Identifier	Output obtained
2^{10}	DEBUG_CODE	dump of retrospective trace
2^{11}	DEBUG_DATA	dump of stack and globals
2^{12}	DEBUG_GRS	dump of general registers
2^{13}	DEBUG_SRS	dump of special registers
2^{14}	DEBUG_KRS	dump of kernel registers
2^{15}	DEBUG_DRS	dump of floating-point registers
FFC0	DEBUG_ALL	all of the above